P802.11i/D3.0, November 2002
DS5909

P802.11i
Draft Supplement to Standard for Telecommunications and Information Exchange Between Systems— LAN/MAN Specific Requirements—

Part 11: Wireless Medium Access Control (MAC) and physical layer (PHY) specifications:

Specification for Enhanced Security

Prepared by the LAN MAN Standards Committee
of the
IEEE Computer Society

Draft Supplement to STANDARD FOR Telecommunications and Information Exchange Between Systems - LAN/MAN Specific Requirements -

Part 11: Wireless Medium Access Control (MAC) and physical layer (PHY) specifications:

Specification for Enhanced Security

Sponsored by the
IEEE 802 Committee
of the
IEEE Computer Society

IEEE Standards Department
Copyright and Permissions
445 Hoes Lane, P.O. Box 1331
Piscataway, NJ 08855-1331, USA

Introduction

(This introduction is not part of IEEE P802.11i, Draft Supplement to STANDARD FOR Telecommunications and Information Exchange Between Systems -LAN/MAN Specific Requirements - Part 11: Wireless Medium Access Control (MAC) and physical layer (PHY) specifications: Specification for Operation in Additional Regulatory Domains)

To be added later

Example:

At the time this supplement to the standard was submitted to Sponsor Ballot, the working group had the following membership:

Stuart J. Kerry, *Chair*
Al Petrick and **Harry Worstell,** *Vice Chairs*
Tim Godfrey, *Secretary*

Dave Halasz, *Chair Task Group i*
Jesse Walker, *Editor, 802.11i*

This list to be added upon conclusion of the sponsor ballot.

Major contributions were received from the following individuals:

Bernard Aboba
Aleg Alimian
Keith Amann
Merwyn Andrade
Arun Ayyagari
Butch Anton
Bob Beach
Simon Black
Simon Blake-Wilson
Nancy Cam-Winget
Clint Chaplin
Greg Chesson
Alan Chickinsky
Frank Ciotti
Donald Eastlake III
Jon Edney
Niels Ferguson
Aaron Friedman
Craig Goston
Larry Green
Dave Halasz
Dan Harkins
Dan Hassett
Russ Housley
Jin-Meng Ho
Dick Hubbard
Hong Jaing
David Johnston
Asa Kalvade
Kevin Karcz
Paul Lambert
Marty Lefkowitz
Onno Letanche
Thomas Maufer
Bill McIntosh
Graham Melville
Tim Moore
Leo Monteban
Mike Moreton

Bob Moskowitz
Dave Nelson
Bob O’Hara
Richard Paine
Henry Ptasinski
Ivan Reede
Carlos Rios
Phil Rogaway
Mike Sabin
Dan Simon
Doug Smith
Mike Sordi
Dorothy Stanley
Denis Volpano
Jesse Walker
Doug Whiting
Albert Young
Glen Zorn

The following persons were on the balloting committee: (To be provided by IEEE editor at time of publication.)

Contents

Draft Supplement to STANDARD FOR Telecommunications and Information Exchange Between Systems - LAN/MAN Specific Requirements -

Part 11: Wireless Medium Access Control (MAC) and physical layer (PHY) specifications:

Specification for Enhanced Security

This supplement is based on the current edition of IEEE Std 802.11, 1999 Edition and the IEEE 802.11a and IEEE 802.11b supplements.

NOTE—The editing instructions contained in this supplement define how to merge the material contained herein into the existing base standard to form the new comprehensive standard as created by the addition of IEEE Std 802.11-1999.

The editing instructions are shown in ***bold itali***c. Three editing instructions are used: change, delete, and insert. ***Change*** is used to make small corrections in existing text or tables. The editing instruction specifies the location of the change and describes what is being changed either by using strikethrough (to remove old material) or underscore (to add new material). ***Delete*** removes existing material. ***Insert*** adds new material with-out disturbing the existing material. Insertions may require renumbering. If so, renumbering instructions are given in the editing instruction. Editorial notes will not be carried over into future editions.

2. Normative references

Add the following text to clause 2:

FIPS PUB 180-1, Secure Hash Standard, April 1995

FIPS PUB 197, Advanced Encryption Standard (AES), 2001 November 26H. Krawczyk, et al, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.

R. Housley, “Advance Encryption Standard (AES) Key Wrap Algorithm,” RFC 3394, September 2002

IEEE STD 802.1X, Standards for Local and Metropolitan Area Networks: Port Based Access Control, June 14, 2001

3. Definitions

Add the following text in the appropriate location in clause 3:

Associated Data: Data that is sent as plaintext but still is to be cryptographically protected in an IEEE 802.11 MSDU. This typically consists of information from the IEEE 802.11 header itself.

Authentication Server: See the IEEE 802.1X specification for a definition of this concept.

Authentication Suite: a set of authentication and key management suite selectors.

Authenticator: See the IEEE 802.1X specification for a definition of this concept.

Authorized: to be explicitly allowed.

Big-Endian: The representation of an integer, with its most significant bit first, least significant bit last, and bytes ordered from most significant to least significant.

Cipher Suite: a set of one or more algorithms, designed to pro provide data privacy, data authenticity or integrity, and/or replay protection.

Controlled Port: An IEEE 802.1X concept, referring to an IEEE 802.1X Port. See IEEE 802.1X for this concept.

Counter-CBC-MAC Mode: a symmetric key block cipher mode providing both privacy using Counter mode and data origin authenticity using CBC-MAC.

Decapsulate: a verb meaning to recover an unprotected packet from a protected one.

Decapsulation: a noun referring to the plaintext data produced by decapsulating an encapsulation.

EAPOL-Key Encryption Key: Key used to encrypt the Key Material field in an EAPOL-Key Message.

EAPOL-Key Key: Combination of EAPOL-Key Encryption key and EAPOL-Key MIC Key.

EAPOL-Key MIC Key: A key used to integrity check an EAPOL-Key Message.

Encapsulate: a verb meaning to construct a protected packet from an unprotected packet.

Encapsulation: a noun meaning the cryptographic payload constructed from plaintext data. This is comprised by the ciphertext, as well as any associated cryptographic state required by the receiver of the data, such as initialization vectors, sequence numbers, message integrity codes, key identifiers, *etc.*

Group: the entities in a wireless network; an AP and associated STAs, or all the STAs in an IBSS network.

Group Master Key: the key that is used as one of the inputs to the Pseudo-Random Function to derive the Group Transient Key.

Group Nonce: A nonce used to derive a Group Transient Key.

Group Transient Key: a value that is derived from the Pseudo-Random Function using the Group Nonce, and is split up into as many as three keys (Temporal Encryption Key, two Temporal MIC Keys) for use by the rest of the system.

Key Counter: a 256 bit (32 octets) counter that is used in the Pseudo-Random Function as a nonce to derive Transient Session Keys. There is a single Key Counter per STA (AP or STA) that is global to that station across all key hierarchies that it is the Key Owner for.

Key Management Service: A service to distribute and manage cryptographic keys within an Robust Security Network

Little-Endian: The representation of an integer, with its least significant bit first, most significant bit last, and bytes ordered from least significant to most significant.

Message Integrity Code: A cryptographic digest, designed to make it computationally infeasible for an adversary to alter data. This is usually called a Message Authentication Code, or MAC, in the literature, but the acronym MAC is already reserved for another meaning in this standard.

Michael: Message Integrity Code for the Temporal Key Integrity Protocol.

Nonce: a value that is never reused with a key. “Never reused within a context” means exactly that, including over all re-initializations of the system through all time.

Offset Codebook Mode: a symmetric key block cipher mode that provides both privacy and data origin authenticity through the use of offset.

Pairwise: two entities that is associated with each other; an AP and one associated station, or a pair of stations in an IBSS network, used to describe the key hierarchies for keys that are shared only between the two entities in a pairwise.

Pairwise Master Key (PMK): the key that is generated on a per-session basis and is used as one of the inputs into the PRF to derive the Pairwise Transient Keys (PTK). For EAP-TLS authentication, the Pairwise Master Key is the key from the RADIUS MS-MPPE-Recv-Key attribute. For Pre-Shared Key authentication, the Pairwise Master Key is the Pre-Shared Key.

Pairwise Transient Key (PTK): a value that is derived from the PRF using the SNonce, and is split up into as many as five keys (Temporal Encryption Key, two Temporal MIC Keys, EAPOL-Key Encryption Key, EAPOL-Key MIC Key) for use by the rest of the system.

Pass phrase: A secret text string supposedly known only by a particular user, employed to prove the user’s identity.

Per-Packet Encryption Key. A unique encryption key constructed for each MPDU, employed by IEEE 802.11 RC4-based protocols.

Per-Packet Sequence Counter: For TKIP, the counter that is used as the nonce in the derivation of the Per-Packet Encryption Key; for AES-based protocols, the Per-Packet IV.

Pre-Shared Key: A static key that is distributed to the units in the system by out-of-band means.

Pseudo-Random Function: a function that hashes various inputs to derive a pseudorandom value. To add liveness to the pseudo random value, a nonce should be one of the inputs; in our case the Key Counter provides nonce.

Robust Security Network: An IEEE 802.11 LAN relying on IEEE 802.1X for its authentication and key management services and CCMP, WRAP, or TKIP for data protection.

Selector: an item specifying a list constituent in an IEEE 802.11 Management Message Information Element.

Supplicant: an IEEE 802.1X concept, which in the context of IEEE 802.11 represents a STA seeking to attach to an IEEE 802 LAN via an IEEE 802.1X Port. See the IEEE 802.1X specification for a complete definition

Temporal Encryption Key: The portion of a transient key used directly or indirectly to encrypt data in packets.

Temporal Key: Combination of temporal encryption key and temporal MIC key.

Temporal MIC Key: The portion of a transient key used to insure the integrity of data packets.

Uncontrolled Port: An IEEE 802.1X concept, referring to an IEEE 802.1X Port. See the IEEE 802.1X specification for a complete definition

4. Abbreviations and acronyms

Add the following text in the appropriate location in clause 4:

AA Authenticator Address

AES Advanced Encryption Standard

AKMP Authenticated Key Management Protocol

ANonce Authenticator Nonce

AS Authentication Server

CBC Cipher-Block Chaining

CBC-MAC CBC Message Authentication Code.

CCM Counter mode with CBC-MAC

CCMP CCM Protocol

CTR Counter mode

EAP Extensible Authentication Protocol (RFC 2284)

EAPOL EAP over LAN (IEEE 802.1X)

EAP-TLS EAP Transport Layer Security (RFC 2716)

GMK Group Master Key

GNonce Group Nonce

GTK Group Transient Key

IETF Internet Engineering Task Force

MIC Message Integrity Code. Because of the special meaning of MAC within the IEEE 802 architecture, this specification uses MIC in place of the standard acronym MAC, which ordinarily stands for Message Authentication Code.

NIST National Institute of Standards and Technologies

NTP Network Time Protocol

OCB Offset Codebook Block mode

OUI Organizationally Unique Identifier

PEAP Protected EAP

PN Packet Number

PRNG Pseudo Random Number Generator

RSN Robust Security Network

RSN IE Robust Security Network Information Element

SNonce Supplicant Nonce

TLS Transport Layer Security (RFC 2246)

TK Temporal Key

TKIP Temporal Key Integrity Protocol

TSC TKIP Sequence Counter

TSN Transition Security Network

TTAK TKIP mixed Transmit Address and Key

WRAP Wireless Robust Authenticated Protocol

5.1.1.4 Interaction with other IEEE 802 layers

Add the following paragraph at the end of clause "5.1.1.4 Interaction with other IEEE 802 layers":

A ***Robust Security Network*** (RSN) depends upon IEEE 802.1X to deliver its authentication and key management services. All STAs and APs in an RSN contain an IEEE 802.1X entity that handles many of these services. This document defines how an RSN utilizes IEEE 802.1X to access these services.

A *Transition Security Network* (TSN) is an RSN that also supports unmodified pre-RSN equipment. A TSN is defined only to facilitate migration to an RSN. A TSN is insecure, since the pre-RSN equipment can compromise the larger network.

Add the following clause after clause "5.1.1.4 Interaction with other IEEE 802 layers" but before clause "5.2 Components of the IEEE 802.11 architecture":

5.1.1.5 Interaction with non-802 Protocols

An RSN utilizes non-802 protocols for its authentication and key management services. These protocols are defined by other standards organizations, such as the IETF.

Add the following clause before clause "5.2.3 Area concepts" and after clause "5.2.2.1 Extended service set (ESS): The large coverage network", renumbering the Figures as appropriate:

5.2.2.2 The Robust Security Network

A Robust Security Network provides a number of security features to the IEEE 802.11 architecture. These features notably include:

- enhanced authentication mechanisms for both APs and STAs;

- key management algorithms;
- dynamic cryptographic keys; and
- enhanced data encapsulation mechanism, called CCMP and, optionally, TKIP and WRAP.

An RSN makes extensive use of IEEE 802.1X protocols with IEEE 802.11 to provide the authentication and key management. This allows IEEE 802.11 to take advantage of work already done in other standards groups.

An RSN introduces several components into the IEEE 802.11architecture. These components are only present in RSN systems.

The first new component is an ***IEEE 802.1X Port***. IEEE 802.1X Ports are present on all STAs in an RSN. They reside above IEEE 802.11 fragmentation and reassembly layer, and all data traffic that flows through the RSN MAC also passes through the IEEE 802.1X Port. The IEEE 802.1X specification describes the internal structure of the IEEE 802.1X Port.

A second component is the ***Authentication Server*** (AS). The AS is an entity that resides in the DS that participates in the authentication of all STAs (including APs) in the ESS. It may authenticate the elements of the RSN itself—i.e., the STAs and APs—or it may provide material that the RSN elements can use to authenticate each other. The AS communicates with the AA on each STA, enabling the STA to be authenticated to the ESS and *vice versa*. Mutual authentication of both the ESS and the STA is an important goal of the RSN. It is important to note that the AS is a logical entity only; in real implementations it may be integrated into the same physical device as an AP, in order to accommodate low end markets such as the home and SoHo.

5.4.2.2 Association

Add the following paragraph after the second paragraph of clause "5.4.2.2 Association":

Within an RSN this situation is slightly different. In an RSN IEEE 802.1X determines when to allow general data traffic across an IEEE 802.11 link. A single IEEE 802.1X Port maps to one association, and each association maps to an IEEE 802.1X Port. After association, the IEEE 802.11 implementation allows any and all data traffic to pass. The IEEE 802.1X Port, however, blocks general data traffic from passing between the STA and the AP until after an IEEE 802.1X authentication procedure completes. Once IEEE 802.1X authentication completes, IEEE 802.1X unblocks to allow data traffic.

5.4.2.3 Reassociation

Add the following paragraphs to the end of clause "5.4.2.3 Reassociation":

As in the case of Association, an AP in an RSN maps a Reassociation to an IEEE 802.1X Port. Although the 802.1X Ports on the STA and AP allows a IEEE 802.1X protocol to traverse the link, they block other data traffic over the link until the IEEE 802.1X signals it has completed successfully.

5.4.2.4 Disassociation

Add the following paragraphs to the end of clause "5.4.2.4 Disassociation":

Informative Note: Disassociation can terminate an in-progress IEEE 802.1X authentication attempt, as disassociation makes the AP unreachable to the STA and *vice versa*. In particular, the IEEE 802.1X protocol between the STA and the AS will not necessarily complete.

5.4.3 Access and confidentiality control services

Change the sentence of the first paragraph of clause "5.4.3 Access and confidentiality control services" from:

Two services are required for IEEE 802.11 to provide functionality equivalent to that which is inherent to wired LANS.

to:

Change the second paragraph of clause "5.4.3 Access and confidentiality control services" from:

Two services are provided to bring the IEEE 802.11 functionality in line with wired LAN assumptions: authentication and privacy. Authentication is used instead of the wired media physical connection. Privacy is used to provide the confidential aspects of closed wired media.

to:

In a pre-RSN WLAN, two services are provided to bring the IEEE 802.11 functionality in line with wired LAN assumptions: authentication and privacy. Authentication is used instead of the wired media physical connection. Privacy is used to provide the confidential aspects of closed wired media.

An RSN does not directly provide either service. Instead, an RSN uses IEEE 802.1X to provide access control and key distribution, and confidentiality is provided as a side effect of key distribution.

5.4.3.1 Authentication

Change the first sentence of the fourth paragraph of clause "5.4.3.1 Authentication" from:

IEEE 802.11 provides link-level authentication between IEEE 802.11 STAs.

to:

IEEE 802.11 supports link-level authentication between IEEE 802.11 STAs.

Add the following paragraphs between the sixth and seventh paragraphs of clause "5.4.3.1 Authentication":

An RSN-capable IEEE 802.11 network also supports authentication based on IEEE 802.1X. IEEE 802.1X authentication utilizes protocols above the MAC to authenticate STAs and the ESS with one another. IEEE 802.1X allows a number of authentication algorithms to be utilized. The standard does not specify a mandatory-to-implement IEEE 802.1X method. In a pure RSN—that is, one deploying only RSN security mechanisms—only Open System Authentication operates at the MAC sub layer itself. An RSN relies on the IEEE 802.1X framework, both to control MSDU flows and to carry the higher layer authentication protocols. In an RSN, the respective IEEE 802.1X Ports of both Access Points and STAs discard MSDUs before the peer is known to have been authenticated. In this associated but unauthenticated state, the IEEE

802.1X Ports permit only the selected IEEE 802.1X authentication protocol to flow across the IEEE 802.11 association.

Since a STA may encounter multiple ESSes, it is necessary to provide a way for a STA to identify the security domain of each, and to determine the authentication mechanisms each supports. If the ESS is an RSN, a STA can determine the authentication protocols in use though Beacons and Probe Responses. Furthermore, the RSN design provides a means by which a STA can indicate the authentication protocol it intends to use with the ESS. It should be noted that the choice of an acceptable authentication protocol is an issue for both APs and the STAs, since the goal of IEEE 802.1X Authentication is mutual authentication between the ESS and the STA, not just authentication of the STA to an AP. Upon encountering an ESS, a STA determines if the authentication mechanisms—Open System, Shared Key, or IEEE 802.1X—supported by the AP suffice, given its own security requirements. A STA might choose not to associate with a particular ESS/AP for many reasons, among them being that the supported authentication mechanisms cannot achieve mutual authentication, or the ESS may constitute an un-trusted security domain.

5.4.3.2 Deauthentication

Change the text of clause "5.4.3.2 Deauthentication" to:

The Deauthentication service is invoked whenever an existing Open System or Shared Key Authentication is to be terminated. Deauthentication is an SS.

In an ESS RSN, Open System Authentication is required for MAC layer authentication. In this environment, Deauthentication results in any association for the deauthenticated station to be terminated, and also results in the 802.1X controlled port for that station being disabled. The Deauthentication notification is provided to 802.1X via the MAC sub layer.

5.4.3.3 Privacy

Add the following paragraph between the fourth and fifth paragraphs of "5.4.3.3 Privacy":

IEEE 802.11 provides four cryptographic algorithms to protect data traffic. Two are based on the RC4 algorithm defined by RSA, and two are based on the Advanced Encryption Standard (AES). This standard refers to these as *WEP*, as *TKIP*, *WRAP*, and CCMP. A means is provided for stations to select the algorithm to be used for a given association.

Add the following clauses after clause "5.4.3.3 Privacy" but before clause "5.5 Relationship among services":

5.4.3.4 Key distribution

The enhanced privacy, data authentication, and replay protection mechanisms require fresh cryptographic keys. These keys need to be created, distributed, and "aged." IEEE 802.11 supports two key distribution mechanisms. The first is manual key distribution. The second is automatic key distribution, and is available only in an RSN that uses a IEEE 802.1X to provide key distribution services.

5.4.3.5 Data Origin Authentication

The data origin authentication mechanism defines a means by which a station that receives a unicast data frame from another station can ensure that the MSDU actually originated from the station whose MAC address is specified in the source address field of the packet. This feature is necessary since an unauthorized station may transmit packets with a source address that belongs to another station. This mechanism is available only to stations using WRAP and TKIP.

Data origin authenticity is only applicable to unicast traffic.

Note: All known algorithms to provide data origin authentication of multicast/broadcast rely on public key cryptography. Because of their computational cost, these methods are inappropriate for bulk data transfers.

5.4.3.6 Replay Detection

The replay detection mechanism defines a means by which a station that receives a unicast data packet from another station can ensure that the packet is not an unauthorized retransmission of a previously sent packet. This mechanism is available only to stations using CCMP, WRAP and TKIP.

5.6 Differences between ESS and IBSS LANs

Add the following paragraphs at the end of Clause "5.6 Differences between ESS and IBSS LANs":

In an IBSS each STA must define and implement its own security model, and each STA must trust the other STAs to implement and enforce a security model compatible with its own. In an ESS the AP enforces a uniform security model.

In an ESS the STA initiates all associations. In an IBSS a STA must be prepared for other STAs to initiate communications. Thus, a STA in an IBSS can negotiate the security algorithms it desires to use when it accepts an association initiated by another station, while in an ESS the AP always chooses the security suite being used.

In an RSN ESS, the AP offloads the authentication decision to an authentication server, while in an IBSS each STA must make its own authentication decision regarding each peer. There is no architectural difference between the two, as in the IBSS case, every STA implements its own Authentication Server.

5.7.6 Authentication

Change the first paragraph in Clause "5.7.6 Authentication" from:

For a STA to authenticate with another STA, the authentication service causes one or more authentication management frames to be exchanged. The exact sequence of frames and their content is dependent on the authentication scheme invoked. For all authentication schemes, the authentication algorithm is identified within the management frame body.

to:

For a STA to authenticate with another STA using either Open System or Shared Key authentication, the authentication service causes one or more authentication management frames to be exchanged. The exact sequence of frames and their content is dependent on the authentication scheme invoked. For both of these authentication schemes, the authentication algorithm is identified within the management frame body.

5.7.7 Deauthentication

Change the first paragraph in Clause "5.7.7 Deauthentication" from:

For a STA to invalidate an active authentication, the following message is sent:

to:

For a STA to invalidate an active authentication that was established using Open System or Shared Key authentication, the following message is sent:

Add the following clauses after Clause "5.8 Reference model", renumbering the Figures as appropriate:

5.9 IEEE 802.11 and IEEE 802.1X

An RSN relies on an IEEE 802.1X entity above IEEE 802.11 to provide authentication and key management services. With this model, decisions as to which packets are permitted onto the DS are made by the IEEE 802.1X entity in addition to the IEEE 802.11 MAC entity.

Given the key role of IEEE 802.1X, a brief summary of IEEE 802.1X and its use with IEEE 802.11 is presented here.

5.9.1 IEEE 802.1X (Informative)

Devices that attach to a LAN, referred to as Systems, have one or more points of attachment to the LAN, referred to as Ports.

The Ports of a System provide the means whereby the System can access services offered by other Systems reachable via the LAN, and also provide the means whereby it can export services to other Systems reachable via the LAN. Port based network access control allows the operation of a System's Port(s) to be controlled in order to ensure that access to its services is only permitted by Systems that are authorized to do so.

For the purposes of describing the operation of Port based access control, a Port of a System is able to adopt one of two distinct roles within an access control interaction:

a) Authenticator. The Port configured to enforce authentication and authorization before allowing access to services that are accessible via that Port adopts the Authenticator role;

b) Supplicant. The Port configured to access the services offered by the Authenticator's system adopts the Supplicant role.

A further System role is described:

c) Authentication Server. The Authentication Server performs the authentication function necessary to check the credentials of the Supplicant on behalf of the Authenticator, and indicates whether or not the Supplicant is authorized to access the Authenticator's services.

As can be seen from these descriptions, all three roles are necessary in order to complete an authentication exchange. A given System can be capable of adopting one or more of these roles; for example, an Authenticator and an Authentication Server can be co-located within the same System, allowing that System to perform the authentication function without the need for communication with an external server. Similarly, a Port can adopt the Supplicant role in some authentication exchanges, and the Authenticator role in others. An example of the latter might occur when a STA acts in the role of a Supplicant in a BSS, but as either the Supplicant or the Authenticator in an IBSS.

A Port Access Entity (PAE) operates the Algorithms and Protocols associated with the authentication mechanisms for a given Port of the System.

In the Supplicant role, the PAE is responsible for responding to requests from an Authenticator for information that will establish its credentials. The PAE that performs the Supplicant role in an authentication exchange is known as the Supplicant PAE.

In the Authenticator role, the PAE is responsible for communication with the Supplicant, and for submitting the information received from the Supplicant to a suitable Authentication Server in order for the credentials to be checked, and for the consequent authorization state to be determined. The PAE that performs the Authenticator role in an authentication exchange is known as the Authenticator PAE.

The Authenticator PAE controls the authorized/unauthorized state of its controlled Port depending upon the outcome of the authentication process.

Figure 1 illustrates that the operation of Port based access control has the effect of creating two distinct points of access to the Authenticator System's point of attachment to the LAN. One point of access allows the uncontrolled exchange of PDUs between the System and other Systems on the LAN, regardless of the authorization state (the ***uncontrolled Port***); the other point of access allows the exchange of PDUs only if the current state of the Port is Authorized (the ***controlled Port***). The uncontrolled and controlled Ports are considered to be part of the same point of attachment to the LAN; any frame received on the physical Port is made available at both the controlled and uncontrolled port, subject to the authorization state associated with the controlled Port.

The point of attachment to the LAN can be provided by any physical or logical Port that can provide a one-to-one connection to a Supplicant System. For example, the point of attachment could be provided by a single LAN MAC in a switched LAN infrastructure. In LAN environments where the MAC method allows the possibility of a one-to-many relationship between an Authenticator and a Supplicant (for example, in shared media environments), the creation of a distinct association between a single Supplicant and a single Authenticator is a necessary pre-condition in order for the access control mechanisms described in this standard to function. An example of such an association would be an IEEE 802.11 association between a station and an access point.

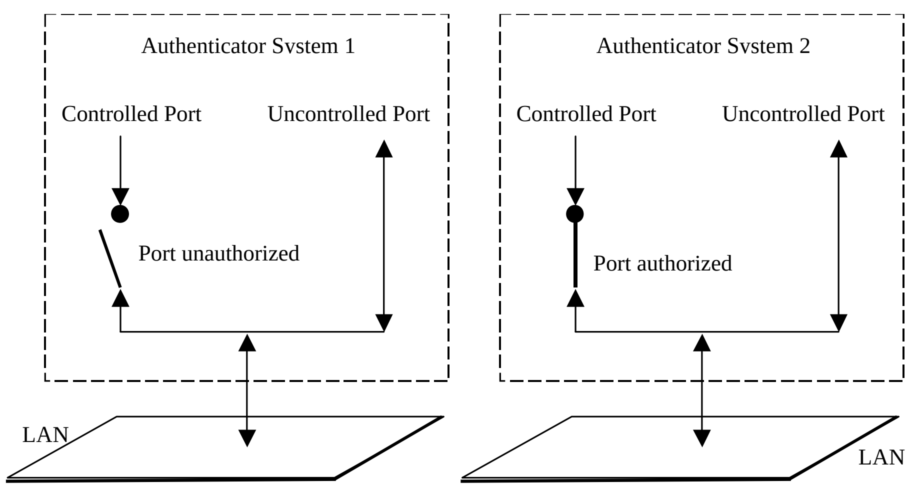


Figure 13

Figure 1 – Uncontrolled and controlled Ports

5.9.2 IEEE 802.11 usage of IEEE 802.1X

IEEE 802.11 depends upon IEEE 802.1X to control the flow of MSDUs between the DS and unauthorized stations by use of the controlled/uncontrolled port model outlined above. EAP authentication packets (contained in IEEE 802.11 MAC data frames) are passed via the IEEE 802.1X authenticator. Non-authentication packets are passed (or blocked) via the controlled port. Each association between a pair of stations creates a unique IEEE 802.1X "port," and authentication takes place relative to that port alone.

IEEE 802.11 depends upon IEEE 802.1X to change its cryptographic keys. IEEE 802.1X may choose to change the keys for a variety of reasons. Some of the reasons include elapsed time or when a certain number of packets have been transmitted or received.

5.9.3 Model description

The following authentication and key management operations are carried out when an IEEE 802.1X Authentication Server is used:

1. The Authenticator and Authentication Server authenticate each other and create a secure channel between them (the possibilities include RADIUS, IPsec, TLS). The security of the channel between the Authenticator and the Authentication Server is outside the scope of this specification.

2. The Supplicant and Authentication Server authenticate each other (e.g., possibilities include EAP-TLS and PEAP) and must generate a Master Key. The authentication must be carried over the Authenticator/Authentication Server secure channel. In addition, there must be crypto-separation over the Authenticator/Authentication Server secure channel for each Supplicant.

3. A Pairwise Master Key (PMK) is generated for use between the Supplicant and Authenticator. The PMK is generated from the EAP master key that is obtained from the Supplicant/Authentication Server authentication. .

4. A 4-way handshake utilizing EAPOL-Key messages occurs between the Supplicant and Authenticator to

    a. Confirm the existence of the PMK;

    b. Confirm that the PMK is current;

    c. Derive the Pairwise Transient Key from the PMK;

    d. Install the encryption and integrity keys into IEEE 802.11;

    e. Confirm the installation of the keys.

5. The Group Transient Key is sent from the Authenticator to the Supplicant to allow the Supplicants to receive, and in an IBSS, transmit broadcast messages, and optionally to transmit and receive unicast packets. EAPOL-Key messages are used to carry out this exchange.

When a Pre-shared Key is used,

1. A Pairwise master key (PMK) is generated for use between the Supplicant and Authenticator. The PMK is the Pre-Shared Key in this case.

2. The 4-way handshake using EAPOL-Key messages is used just as in the Authentication Server case.

3. The Group Transient Key is sent from the Authenticator to the Supplicant just as in the Authentication Server case.

There are two implementations of this architecture:

1. For an ESS, the AP is the Authenticator, and associated STAs are the Supplicants. The Authentication Server may be a RADIUS Server.

2. For an IBSS, each STA is an Authenticator and Supplicant. Each STA implements an Authentication Server, or else uses a Global pre-shared key is required.

5.9.3.1 Frame exchange overview

Before IEEE 802.11 can protect packets, the STA must perform IEEE 802.11 Open System Authentication and associate to the AP. These steps allow the STA and AP to negotiate security association characteristics, including the authenticated key management, unicast and multicast cipher suites employed. IEEE 802.11 Open System Authentication and association are used to retain legacy IEEE 802.11 state flow. Figure 2 depicts how a STA discovers an AP and negotiates a security policy.

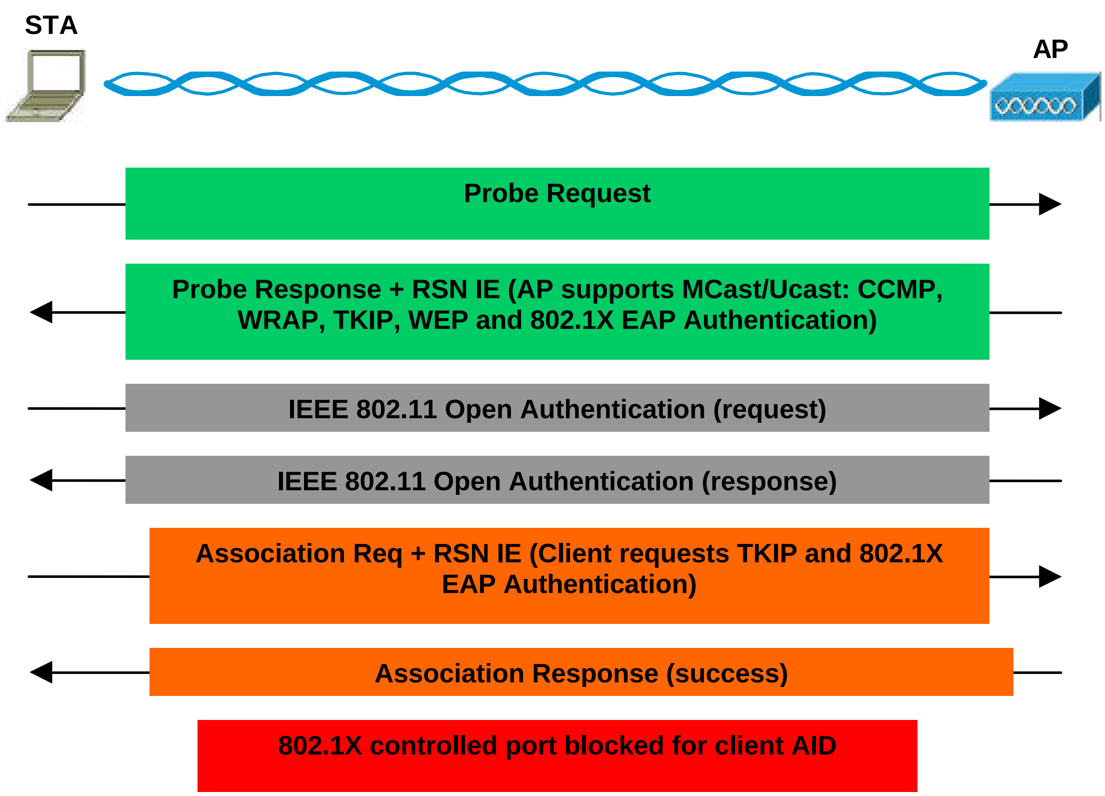


Figure 2—Establishing the IEEE 802.11 connection and negotiation

Once the STA and AP successfully establish a common security policy, both filter both data traffic, restricting this to IEEE 802.1X EAP authentication frames. In the next phase the STA to successfully authenticate with an Authentication Server (AS), as depicted by Figure 3.

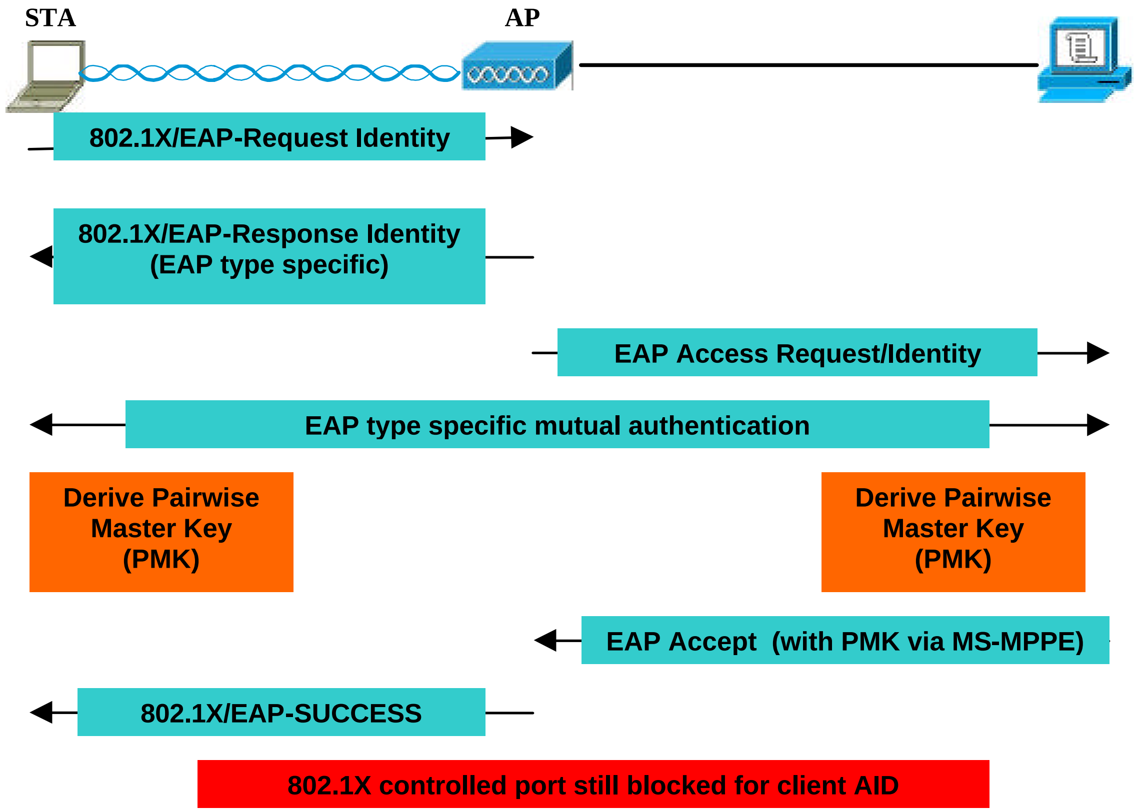


Figure 3—IEEE 802.1X EAP authentication

In order for the STA to avoid rogue APs and the AP unauthorized STAs, the STA and AP must mutually authenticate and prove the communication is live and not being replayed. Both the AP and the STA is still block general IEEE 802.11 data packets during this phase, allowing only IEEE 802.1X EAP packets to flow, The IEEE 802.1X authentication step achieves mutual authentication with the STA and derives fresh, never-before-used per-link keys, which are required to protect traffic over the association. The per-link key, known as a ***Pairwise Transient Key*** (PTK), is achieved through a protocol called the ***4-way handshake***, depicted in Figure 4. Clause 8.5 describes the 4-way handshake in greater detail.

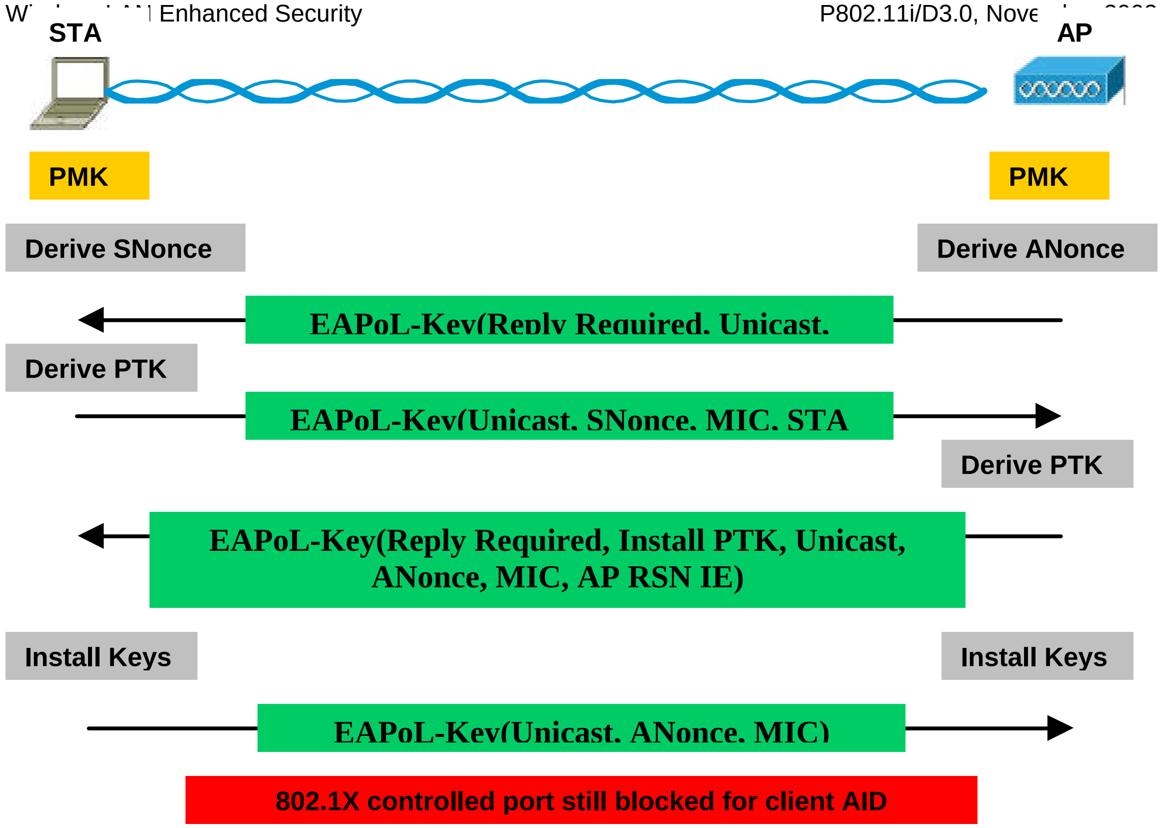


Figure 4—Establishing pairwise keys

Once the STA and AP have authentication and established a fresh pairwise key, the AP can use it to deliver the key required to protect multicast traffic, the ***Group Transient Key*** (GTK). This last phase is achieved with a two message exchange, called the ***Group Key Handshake***. Upon its success, both STA and AP open the IEEE 802.1X port and allow communication over a protected channel. The last phase is shown in Figure 5.

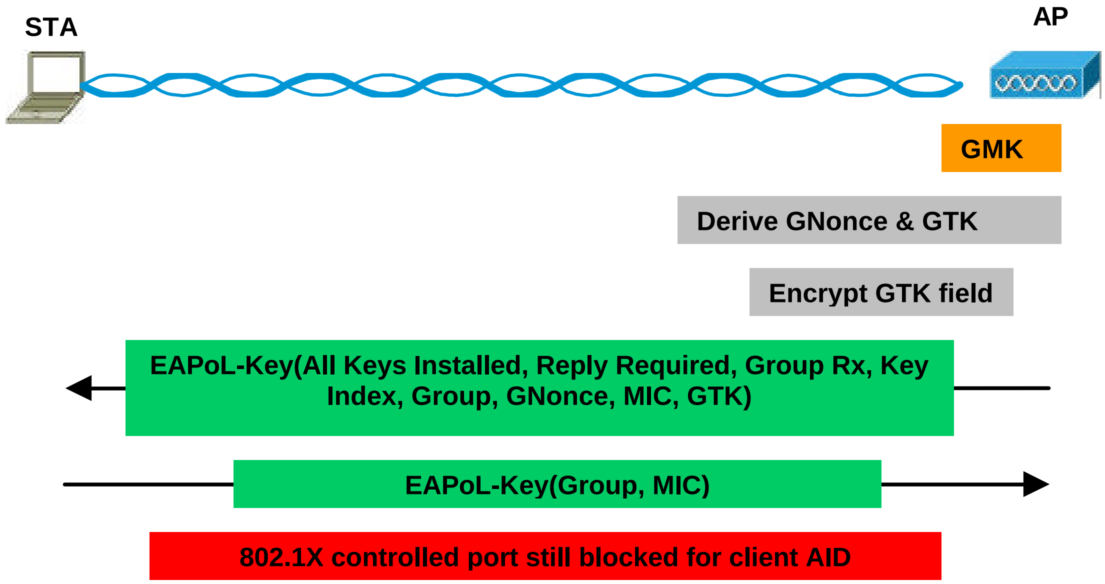


Figure 5—Group key delivery - final phase

5.9.4 Deployment discussion

The Authenticator/Authentication Server authentication protocol is out of scope, but, to provide security assurances, the protocol needs the following characteristics:

1. Authenticate the Authenticator and Authentication Server.
2. Provide a secure channel for the Supplicant/Authentication Server authentication and provide separation of different Supplicant to Authentication Server exchanges.
3. Pass the generated key from the Authentication Server to the Authenticator for use by the Authenticator to communicate to the Supplicant.

Suitable protocols include RADIUS and Diameter.

Change the phrase "Wired Equivalent Privacy (WEP)" in Clause 7.1.3.1 to "Protected Frame".

Change "WEP" in Figure 13 to "Protected Frame".

Change the title of Clause 7.1.3.1.9 to:

7.1.3.1.9 Protected Frame field

Change the text of Clause 7.1.3.1.9 to:

The Protected Frame field is one bit in length. The Protected Frame field is set to 1 if the Frame Body field contains information that has been processed by a cryptographic encapsulation algorithm. The Protected Frame field is only set to 1 within frames of Type Data and frames of Type Management, Subtype Authentication. The Protected Frame field is set to 0 in all other frames. When the Protected Frame bit is set to 1, the Frame Body field is protected utilizing the cryptographic algorithm selected during association or Reassociation and expanded as defined in Clause 8.

Change the text of paragraph from Clause 7.2.2 reading

The frame body consists of the MSDU or a fragment thereof, and a WEP IV and ICV (if and only if the WEP subfield in the frame control field is set to 1). The frame body is null (0 octets in length) in data frames of Subtype Null function (no data), CF-Ack (no data), CF-Poll (no data), and CF-Ack+CF-Poll (no data).

to

The frame body consists of the MSDU or a fragment thereof, and a security header and trailer (if and only if the Protected Frame subfield in the frame control field is set to 1). The frame body is null (0 octets in length) in data frames of Subtype Null function (no data), CF-Ack (no data), CF-Poll (no data), and CF-Ack+CF-Poll (no data).

7.2.3.1 Beacon frame format

Add the following rows to the end of Table 4 in Clause "7.2.3.1 Beacon frame format":

14	RSN Information Element	A Beacon may specify a single RSN Information Element.

7.2.3.4 Association Request frame format

Add the following rows to the end of Table 7 in Clause "7.2.3.4 Associate Request frame format":

5	RSN Information Element	An association request may specify a single RSN Information Element.

7.2.3.6 Reassociation Request frame format

Add the following rows to the end of Table 9 in Clause "7.2.3.6 Reassociate Request frame format":

6	RSN Information Element	A Reassociation request may specify a single RSN Information Element.

7.2.3.9 Probe Response frame format

Add the following rows to the end of Table 12 in Clause "7.2.3.9 Probe Response frame format":

10	RSN Information Element	A Probe response may specify a single RSN Information Element.

7.2.3.10 Authentication frame format

Add the following text after the first sentence of Clause "7.2.3.10 Authentication frame format"

Only Open System Authentication frames may be used with RSN.

7.3.1.4 Capability Information field

Add the following paragraphs to Clause 7.3.1.4:

STAs (including APs) that include the RSN IE in beacons and probe responses shall set the Privacy subfield to 1 in any frame that includes it.

Delete the last row and then add the following rows to "Table 18—Reason codes":

13	Invalid Information Element
14	MIC failure
15	4-way handshake timeout
16	Group key update timeout
17	Information element in 4-way handshake different from (Re-)associate request/Probe response/Beacon
18	Multicast Cipher is not valid
19	Unicast Cipher is not valid
20	AKMP is not valid
21	Unsupported RSNE version
22	Invalid RSNE Capabilities
23	IEEE 802.1X Authentication failed
24-65535	Reserved

Add the following row to "Table 20 – Element IDs":

RSN Information Element	48

Add the following clause after Clause "7.3.2.8 Challenge Text element" but prior to Clause "8 Authentication and privacy", renumbering Tables and Figures as appropriate:

7.3.2.17 RSN Information Element (RSN IE)

The RSN Information Element (RSN IE) lists authentication and pairwise key cipher suite selectors, a single group key cipher suite selector, and an RSN capabilities field. All STAs implementing RSN shall support this element.

Element ID	Length	Version	Group Key Cipher Suite	Pairwise Key Cipher Suite Count	Pairwise Key Cipher Suite List	Authenticated Key Management Suite Count	Authenticated Key Management Suite List	RSN Capabilities
1 octet	1 octet	2 octets	4 octets	2 octets	$4 \cdot m$ octets	2 octets	$4 \cdot n$ octets	2 octets

Figure 6—RSN Information Element format

Informative Note. The count fields of the RSN IE were chosen to be two octets each to improve alignment.

All fields use the bit convention from 7.1.1. The RSN IE, if supplied, shall contain up to and including the Version field. The group key cipher suite field, pairwise cipher suite field, authenticated key management suite field, and RSN Capabilities field are optional. If the group key suite field is not supplied, then the pairwise key cipher suite and authenticated key management suite fields shall not be supplied. If the group key cipher suite field is supplied but not the pairwise key suite field, then the authenticated key management suite field shall not be supplied.

Element ID shall be 48 decimal (30 hex).

Length gives the number of octets in the information element.

The Version field indicates the version number of the RSN protocol. The range of Version field values a STA supports shall be contiguous.

RSN Version 1 shall indicate the following:

1. A STA may support IEEE 802.11 Open System Authentication.
2. A STA sets the Privacy bit set in the same way as WEP.
3. A STA supports the RSN IE. An AP supporting RSN shall include the RSN IE in Beacons and Probe Responses. A STA supporting RSN shall include the RSN IE in the Association and Reassociation Requests.
4. A STA supports CCMP.
5. A STA supports key updates using EAPOL-Key descriptor from this document.

A suite selector has the following format:

OUI – 3 Octets	Suite Type – 1 octet

Figure 7—Suite selector format

The order of the OUI field shall follow the ordering convention for MAC addresses from IEEE 802.11 7.1.1.

Table 1 – Authenticated Key Management Suite Selectors

OUI	Value	Meaning	
		Authentication Type	Key Management Type
00:00:00	0	Reserved	Reserved
00:00:00	1	Unspecified authentication over IEEE 802.1X– RSN default	IEEE 802.1X Key Management as defined in 8.5 – RSN default
00:00:00	2	None	IEEE 802.1X Key Management as defined in 8.5, using pre-shared key
00:00:00	3-255	Reserved	Reserved
Vendor Specific	Any	Vendor Specific	Vendor Specific
Other	Any	Reserved	Reserved

The Authenticated Key Management suite selector value 00:00:00:1 "Unspecified authentication over IEEE 802.1X" with "IEEE 802.1X key management as defined in 8.5" shall be the assumed default when the Authenticated Key Management Suite Selector field is not supplied.

Informative Note. The Selector value 00:00:00:1 specifies only that IEEE 802.1X is used as the authentication transport, and that IEEE 802.1X selects the authentication mechanism.

The Authenticated Key Management suite selector value 00:00:00:2 "Pre-shared key over IEEE 802.1X" is used when a pre-shared key is used with IEEE 802.1X.

Informative Note: The inclusion of different Authentication types allows the simplification of the User Interface. It allows the pre-shared key UI to be enabled/disabled on stations depending on the configuration of the AP so users are only asked for the information that is required for any particular scenario.

Informative Note: This specification defines no vendor specific Authenticated Key Management Suites. The category "Vendor Specific" is reserved as a standardized way to introduce suites.

Table 2 – Cipher Suite Selectors

OUI	Value	Meaning
00:00:00	0	None
00:00:00	1	WEP-40
00:00:00	2	TKIP
00:00:00	3	WRAP
00:00:00	4	CCMP – default in an RSN
00:00:00	5	WEP-104
00:00:00	6-255	Reserved
Vendor OUI	Other	Vendor Specific
Other	Any	Reserved

The cipher suite selector 00:00:00:4 "CCMP" shall be the default cipher suite value.

The cipher suite selector 00:00:00:1 "WEP" is only valid as a cipher suite in a TSN.

Use of CCMP or WRAP as the group key cipher suite with TKIP or WEP as the pairwise key cipher suite shall not be supported.

The cipher suite selector 00:00:00:0 “None” is only valid as the unicast cipher suite. An AP may specify the selector 00:00:00:0 “None” for a pairwise key cipher suite if it does not support any pairwise cipher suites. An AP shall not specify the selector 00:00:00:0 “None” as the group key cipher suite selector. The group key cipher suite selector in the Associate Request and the Reassociate Request shall match the value the STA received in the Probe Response or the Beacon.

Informative Note: The selector 00:00:00:0 “None” informs STAs that the AP is not configured to support pairwise key cipher suites.

Informative Note: This specification defines no vendor specific Cipher Suites. The category “Vendor Specific” is reserved as a standardized way to introduce suites.

It does not make sense to use every cipher suite in any context. Table 3 indicates the circumstances under which each may be used.

Table 3—Cipher Suite Usage

Cipher Suite Selector	**Group Key, IBSS**	**Group Key, ESS**	**Pairwise Key**
None	No	No	Yes
WEP	No	Yes	No
TKIP	Yes	Yes	Yes
WRAP/CCMP	Yes	Yes	Yes

The RSN Capability Information field indicates requested or advertised capabilities. The length of the RSN Capability Information field is two octets. An AP sets the Pre-authentication Subfield (Bit 0) of the RSN Capability Information field to signal it supports Pre-Authentication, and it clears the subfield when it does not support Pre-Authentication. A STA sets the Pairwise Key Subfield to 1 if the STA supports Pairwise keys using default keys rather than using key-mapping keys, and clears the subfield otherwise. The remaining subfields of the RSN Capability Information field are reserved and shall be set to zero on transmission and ignored on reception. The value of the capability information field shall be taken as 0 if the field is not available in the RSN information element. The format of the Capability Information field is as illustrated in Figure 8.

Figure 8—RSN Capabilities

The TKIP Number of Replay Counters contains the value of dot11TKIPNumberOfReplayCounters. See Section 8.3.2.2.4. If the field does not exist in the information element then the value of 0 is to be assumed. The meaning of dot11TKIPNumberOfReplayCounters is:

0 1 replay counters

1 2 replay counters

2 4 replay counters,

3 16 replay counters

Informative Note. If a security policy does not allow particular cipher or authentication suites, then APs and STAs should be configured to not advertise or select these suites in the RSN IE

Informative Note: The following represent example information elements:

1. 802.1X authentication, CCMP pairwise and group key cipher suites (WEP and TKIP not allowed).
    30, // information element id, 48 expressed as Hex value
    14, // length in octets, 20 expressed as Hex value
    01 00, // Version 1
    00 00 00 04, // CCMP as group key cipher suite
    01 00, // pairwise key cipher suite count
    00 00 00 04, // CCMP as pairwise key cipher suite
    01 00, // authentication count
    00 00 00 01 // 802.1X authentication
    00 00 // No capabilities

2.
    30, // information element id, 48 expressed as Hex value
    14, // length in octets, 20 expressed as Hex value
    01 00, // Version 1
    00 00 00 04, // CCMP as group key cipher suite
    01 00, // pairwise key cipher suite count
    00 00 00 04, // CCMP as pairwise key cipher suite
    01 00, // authentication count
    00 00 00 01 // 802.1X authentication
    80 00 // No capabilities

3.
    30, // information element id, 48 expressed as Hex value
    12, // length in octets, 20 expressed as Hex value
    01 00, // Version 1
    00 00 00 01, // WEP as group key cipher suite
    01 00, // pairwise key cipher suite count
    00 00 00 00, // No pairwise key cipher suite
    01 00, // authentication count
    00 00 00 01 // 802.1X authentication

Replace Clause 8 "Authentication and Privacy" with the following text:

8 Security

8.1 Framework

This standard defines two classes of security algorithms for IEEE 802.11 networks: *pre-RSN security* algorithms, and algorithms for a Robust Security Network, called *RSN security* algorithms. Equipment implementing Robust Security Network algorithms are called *RSN-capable*, while earlier IEEE 802.11 equipment are called *pre-RSN equipment*. It also supports combinations of RSN and pre-RSN equipment in the same WLAN. Such a network is called a *Transition Security Network*, or *TSN*, to emphasize the transitional nature of such combinations.

> ***<u>Important Informative Security Warning</u>. Transition means just that. A TSN cannot provide the assurances of an RSN. Compromise of communication between pre-RSN and RSN equipment can compromise communication strictly among RSN equipment. Organizations mixing RSN and pre-RSN equipment should be encouraged to migrate to homogeneous RSN networks as rapidly as is feasible.***

All security algorithms are optional, but all IEEE 802.11 implementations claiming security shall implement the mandatory RSN components.

8.1.1 Security components

Pre-RSN security consists of two basic subsystems:

- WEP privacy, to encapsulate data, and
- IEEE 802.11 authentication.

8.2.2.1 describes WEP, while 8.2.3.1 describes the IEEE 802.11 authentication procedures.

RSN security consists of two basic subsystems:

- Data privacy mechanism:
  - TKIP, to provide minimally adequate level of data privacy for pre-RSN hardware conforming to the 1999 issue of this standard;
  - WRAP, an optional AES-based protocol, to provide robust data privacy for the long term; and
  - CCMP, another AES-based protocol, to provide robust data privacy. Any implementation claiming to provide security shall implement CCMP
- Security association management:
  - RSN negotiation procedures, to establish a security context;
  - IEEE 802.1X authentication, replacing IEEE 802.11 authentication;
  - IEEE 802.1X key management, to provide cryptographic keys;

8.1.2 Identifying pre-RSN equipment

Pre-RSN devices conform to the 1999 issue of this standard. These devices do not include the RSN IE in their Beacons and Probe Responses, and in Association and Reassociation Requests. Pre-RSN devices ignore the presence or otherwise of the RSN IE in received messages.

8.1.3 Identifying RSN-capable equipment

An RSN-capable AP shall, and a non-AP STA may, include the RSN IE in all Beacons, Probe Responses, Association Requests, and Reassociation Requests. When included, this IE advertises the sender as RSN-capable. Including the RSN IE shall be the default for RSN-capable non-AP STAs.

An RSN-capable STA may identify another RSN-capable STA by noting that the RSN IE is included in any Beacon, Probe Response, Association Request, or Reassociation Request it receives from the peer. An RSN-capable STA may identify Pre-RSN equipment by the peer's failure to include the RSN IE.

> Informative Note: There is no requirement for a non-AP STA to always include the RSN IE in all the association establishment messages. For example, if a STA migrates to an unknown ESS in a new security domain, it may not be able to communicate because it has not been issued the appropriate credentials. This forces the association, if accepted, to fall back to use pre-RSN security mechanisms only. The responding peer STA is not required to accept the association request in this instance, as doing so may violate its own security policy. Every RSN-capable AP shall include the RSN IE to participate in RSN security.

8.1.4 Mixtures of RSN and pre-RSN equipment

An RSN-capable AP in an ESS or a STA in an IBSS may communicate with both RSN-capable and pre-RSN equipment simultaneously. An RSN-capable STA in an ESS may communicate with either RSN-capable or legacy APs, but shall not do so simultaneously. These rules permit migration from deployments based on legacy WEP security to RSN-based security.

8.1.5 Operation

RSN supports two models of operation. One model is based on IEEE 802.1X authentication, while the other depends on a global pre-shared key.

RSN-capable STAs use Beacons and Probe request to identify other RSN-capable peer STAs. When the peer indicates it is RSN-capable, the STA shall implement the following sequence of procedures in the IEEE 802.1X authentication model:

1. First it associates and negotiates the security parameters used with the association. 8.4.2 and 8.4.3 describe the RSN negotiation procedures.

2. Next it authenticates, using the agreed upon association mechanism. 8.4.6 and 8.4.7 describe the IEEE 802.11 use of IEEE 802.1X authentication.

3. Third, it executes a key exchange algorithm, based on the IEEE 802.1X EAPOL Rekey protocol. Clause 8.5 describes the IEEE 802.11 use of IEEE 802.1X key management, to obtain temporal keys.

4. Finally, it uses the agreed upon temporal keys and cipher suites to protect the link. 8.3.2, 8.3.3, and 8.3.4 describe the three defined data RSN encapsulation mechanisms.

If the peer fails to indicate it is RSN-capable, the STA may fall back to the following procedures:

1. first uses 1999 IEEE 802.11 (Pre-RSN) authentication;

2. followed by association;

3. optionally followed by use of legacy WEP.

8.2.3.1 describes pre-RSN authentication, while 8.2.2.1 describes WEP.

If the BSS is based on a global pre-shared key, the STA instead executes the following sequence of procedures:

1. It runs the Clause 8.5 the key exchange, to establish pairwise and group keys and cipher suites. It uses the global pre-shared key as the pairwise master key for each such exchange

2. It uses the established keys and cipher suites to protect the link.

8.1.6 RSN assumptions and constraints

RSN assumes:

1. Mutual authentication of the IEEE 802.1X AS and the STA. This assumption is intrinsic to IEEE 802.11 LANs and cannot be removed without compromising security.

2. In particular, the mutual authentication requirement implies an unspecified prior enrollment process, as the STA must be able to identify the ESS or IBSS as an entity that it regards as genuinely trustworthy. The non-secured IEEE 802.11 model of promiscuous roaming does not and cannot provide security in a WLAN. This assumption is intrinsic to IEEE 802.11 and cannot be removed without compromising security.

   Informative Note: This assumption complicates some business models, such as those used by IEEE 802.11 hot spot providers, but this in no way eliminates the assumption. Enrollment can be indirect, e.g., an organization might use a PKI for authentication, signing the hot spot provider's certificate with a key STAs from their organization trust. Such a signing key can only be employed for this one purpose—certifying that the bearer's is a party trusted to enforce the signer's security policy—or security of the WLAN is lost. In practice service level agreements and auditing will be needed to be able to verify that the provider actually enforces the security policy delegated in this manner.

3. RSN assumes that either the mutual authentication is strong or is somehow shielded from unauthorized reception. This assumption is intrinsic to IEEE 802.11 LANs and cannot be removed without compromising security.

4. Authentication derives a fresh—i.e., never before used—session key.

5. In an ESS all APs lie entirely within the security boundary surrounding the IEEE 802.1X AS. This is a very strong configuration constraint. In practice this implies that either the IEEE 802.1X server is embedded in the AP, or else the AP is physically secure (e.g., physical access to the AP is controlled; access—both physical and by network—to the DS is controlled; the AP shielded from all unauthorized radio transmissions, etc.), and the communication channel between the AS and the AP lies entirely within the security boundary as well.

6. In an ESS that supports roaming, all channels between any pair of APs through the DS are within the same security boundary. This again is a very strong configuration constraint. It implies that the DS is wired, physically secured, and secured from all outside attacks, including those that might be launched via IEEE 802.1X authentication itself. Thus, RSN cannot support one of the most common home configurations, where the IEEE 802.11 LAN is itself the DS.

7. Key generation of a 256-bit key at the Supplicant and Authentication Server for use by the Supplicant and Authenticator.

8.2 Pre-RSN security methods

Except for Open System Authentication, all pre-RSN security mechanisms have been deprecated, as they fail to meet their security goals. They can be easily compromised. New implementations should support pre-RSN methods only to aid migration to RSN methods.

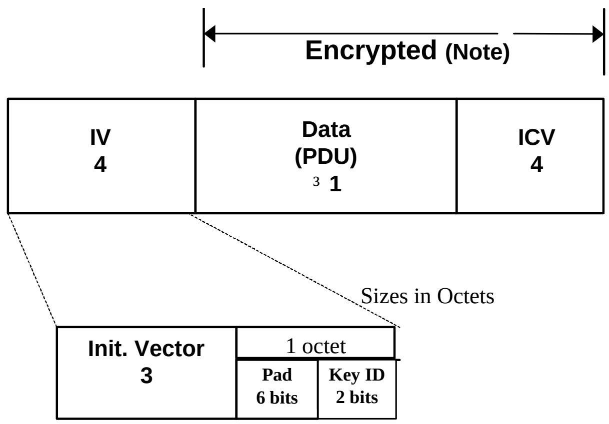
8.2.2 Wired Equivalent Privacy (WEP)

8.2.2.1 WEP overview

WEP was defined in the 1999 issue of this standard as a means of protecting the confidentiality of data exchanged among authorized users of a wireless LAN from casual eavesdropping. Implementation of WEP is optional.

8.2.2.2 WEP MPDU format

Figure 9 depicts the encrypted Frame Body as constructed by the WEP algorithm.



NOTE: The encipherment process has expanded the original MPDU by 8 Octets, 4 for the Initialization Vector (IV) field and 4 for the Integrity Check Value (ICV). The ICV is calculated on the Data field only.

Figure 9—Construction of Expanded WEP MPDU

The WEP ICV shall be a 32-bit field. The expanded Frame Body shall start with a 32-bit IV field. This field shall contain three sub fields: a three-octet field that contains the initialization vector, a 2-bit key ID field, and a 6-bit pad field. The ordering conventions defined in 7.1.1 apply to the IV fields and its sub fields and to the ICV field. The key ID subfield contents select one of four possible secret key values for use in decrypting this Frame Body. Interpretation of these bits is discussed further in 8.2.2.1.4.6. The contents of the pad subfield shall be zero. The key ID occupies the two msb of the last octet of the IV field, while the pad occupies the six lsb of this octet.

8.2.2.3 WEP state

WEP uses encryption keys only; it performs no data authentication, so does not have data integrity keys. WEP(-40) encryption keys shall be 40-bits in length. WEP-104 keys shall be 104-bits in length. WEP uses two types of encryption keys: key-mapping keys and default keys.

A key-mapping key is an unnamed key corresponding to a distinct <TA,RA> pair. Implementations shall use the key-mapping key if it is configured for a <TA,RA> pair. This means the key-mapping key shall be used to WEP encapsulate or decapsulate MPDUs transmitted by TA to RA, regardless of the presence of other key types. When a key-mapping key for an address pair is present, the WEP key ID field in the MPDU shall be set to zero on transmit and ignored on receive.

A default key is an item in a four-element MIB array called *dot11WEPDefaultKeys*, named by the value of a related array index called *dot11WEPDefaultKeyID*. If a key-mapping key is not configured for a WEP MPDU's <TA,RA> pair, WEP shall use a default key to encapsulate or decapsulate it. On transmit the key selected is the element of the *dot11DefaultKeys* array given by the index *dot11WEPDefaultKeyID*—a value of 0, 1, 2, or 3—corresponding to the first, second, third, or fourth element, respectively, of *dot11WEPDefaultKeys*. The value the transmitter encodes in the WEP key ID field of the transmitted MPDU shall be the *dot11WEPDefaultKeyID* value. The receiver shall use the key id field of the MPDU to index into *dot11WEPDefaultKeys* to obtain the correct default key. All WEP implementations shall support default keys.

Informative Note: Many implementations also support 104-bit WEP keys. These are used exactly like 40-bit WEP keys: a 24-bit WEP IV is prepended to the 104-bit key to construct a 128-bit WEP seed, as explained below in 8.2.2.4.3. The resulting 128-bit WEP seed is then consumed by the RC4 stream cipher.

This construction based on 104-bit keys affords no more assurance than the 40-bit construction and its implementation and use is in no way condoned by this standard. Rather, the 104-bit construction is noted only to document *de facto* practice.

This document sometimes refers to 40-bit WEP as WEP-40, and to 104-bit WEP as WEP-104.

The default value for all WEP keys shall be null. WEP implementations shall discard the containing MSDU and generate an MA-UNITDATA-STATUS.indication with transmission status indicating that a frame may not be encapsulated with a null key in response to any request to encapsulate an MPDU with a null key.

8.2.2.4 WEP procedures

8.2.2.4.1 WEP ICV algorithm

The WEP ICV shall be computed using the CRC-32, as defined in 7.1.3.6, calculated over the MPDU Data (PDU) field.

8.2.2.4.2 WEP encryption algorithm

A WEP implementation shall use the RC4 stream cipher from RSA Data Security, Inc., as its encryption and decryption algorithm. RC4 uses a PRNG to generate a *key stream* that it XORs with a plaintext data stream to produce ciphertext or with a ciphertext stream to produce plaintext.

8.2.2.4.3 WEP seed construction

A WEP shall construct a per-packet key, called a *seed*, by concatenating an encryption key to an *initialization vector* (IV).

For WEP(-40), bits 0 through 39 of the WEP key correspond to bits 24 through 63 of the seed, and bits 0 through 23 of the IV correspond to bits 0 through 23 of the seed, respectively. For WEP-104, bits 0 through 103 of the WEP key correspond to bits 24 through 127 of the seed, and bits 0 through 23 of the IV correspond to bits 0 through 23 of the seed, respectively. The bit numbering conventions in 7.1.1 apply to the seed. The seed shall be the input to RC4, in order to encrypt or decrypt the WEP Data and ICV fields.

The WEP implementation encapsulating an MPDU should select a new IV for every packet it WEP encapsulates. The IV selection algorithm is unspecified. The algorithm the encapsulation uses to select the encryption key used to construct the seed is also unspecified.

The WEP implementation decapsulating an MPDU shall use the IV from the received MPDU's Init Vector subfield. Clause 8.2.2.1.4.6 specifies how the decapsulator selects the key to use to construct the per-packet key.

8.2.2.4.4 WEP MPDU encapsulation

WEP shall apply three transformations to the plaintext MPDU to effect the WEP encapsulation. WEP computes the ICV over the plaintext Data and then appends this after the MPDU data. WEP encrypts the MPDU plaintext Data and ICV using RC4 with a seed constructed, as specified in Clause 8.2.2.1.4.3. WEP encodes the IV and key id into the IV field, prepended to the encrypted Data field.

Figure 10 depicts the WEP encapsulation process. The ICV shall be computed and appended to the plaintext data prior to encryption, but the IV encoding step may occur in any order convenient for the implementation.

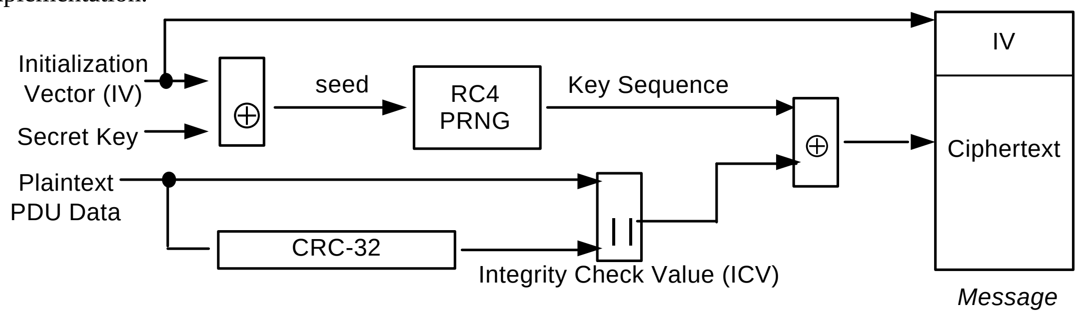


Figure 10—WEP Encapsulation Block Diagram

8.2.2.4.5 WEP MPDU decapsulation

WEP shall apply three transformations to the WEP MPDU to decapsulate its payload. WEP extracts the IV and key id from the received MPDU. The key id identifies the decryption key to use, which is combined as described in Clause 8.2.2.1.4.3 to construct the seed for this MPDU. WEP uses the constructed seed to decrypt the Data field of the WEP MPDU; this produces plaintext data and an ICV. Finally WEP recomputes the ICV and bit-wise compares it with the decrypted ICV from the MPDU. If the two are bit-wise identical, then WEP removes the IV and ICV from the MPDU, which is accepted as valid; if they differ in any bit position, WEP generates an error indication to MAC management. MSDUs with erroneous MPDUs (due to inability to decrypt) shall not be passed to LLC.

Figure 11 depicts a block diagram for WEP decapsulation. Unlike encapsulation, the decapsulation steps shall be in the indicated order.

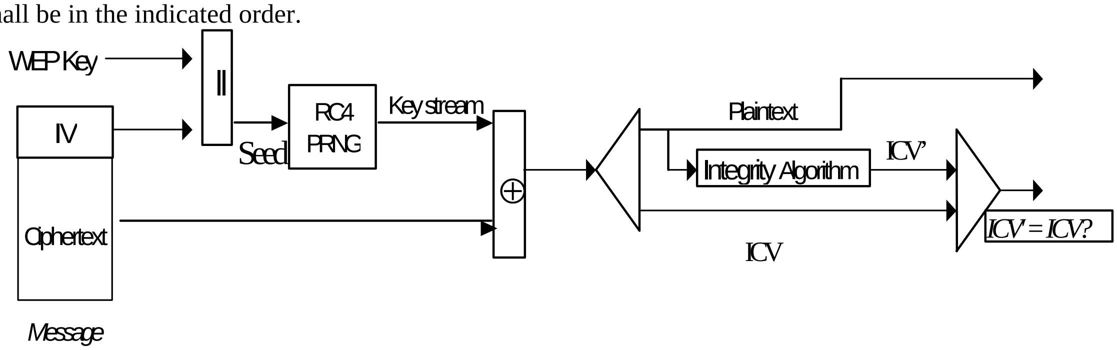


Figure 11—WEP Decapsulation Block Diagram

8.2.2.4.6 WEP MIB attributes

An MPDU of type Data with the WEP subfield of the Frame Control field equal to 1 is called a *WEP MPDU*. Other MPDUs of type Data are called *non-WEP MPDU*s.

A STA shall not transmit WEP encapsulated MPDUs when value of the MIB variable *dot11PrivacyInvoked* is “false.” This MIB variable does not affect MPDU or MMPDU reception.

```
if dot11PrivacyInvoked is “false”
    the MPDU is transmitted without WEP encapsulation
else
    if (the MPDU has an individual RA and
    there is an entry in dot11WEPKeyMappings for that RA)
        if that entry has WEPOn set to “false”
            the MPDU is transmitted without WEP encapsulation
        else
            if that entry contains a key that is null
                discard the MPDU’s entire MSDU and generate an
                MA-UNITDATA-STATUS.indication primitive to notify
                LLC that the MSDU was undeliverable due to a null WEP key
            else
                encrypt the MPDU using that entry’s key, setting the KeyID
                subfield of the IV field to zero
    else
        if (the MPDU has a group RA and the Privacy subfield
        of the Capability Information field in this BSS is set to 0)
            the MPDU is transmitted without WEP encapsulation
        else
            if dot11WEPDefaultKeys[dot11WEPDefaultKeyID] is null
                discard the MPDU’s entire MSDU and generate an
                MA-UNITDATA-STATUS.indication primitive to notify
                LLC that the MSDU was undeliverable due to a null WEP key
            else
                WEP encapsulate the MPDU using the key
                dot11WEPDefaultKeys[dot11WEPDefaultKeyID],
                setting the KeyID subfield of the IV field to
                dot11WEPDefaultKeyID
```

When the boolean attribute aExcludeUnencrypted is set to True, non-WEP MPDUs shall not be indicated at the MAC service interface, and only MSDUs successfully reassembled from successfully decrypted MPDUs shall be indicated at the MAC service interface. When receiving a frame of type Data, the values of *dot11PrivacyOptionImplemented*, *dot11WEPKeyMappings*, *dot11WEPDefaultKeys*, *dot11WEPDefaultKeyID*, and aExcludeUnencrypted in effect at the time the PHY-RXSTART.indication primitive is received by the MAC shall be used according to the following decision tree:

```
if the Protected Frame subfield of the Frame Control Field is zero
    if aExcludeUnencrypted is “true”
        discard the frame body without indication to LLC and increment
        dot11WEPExcludedCount
    else
        receive the frame without WEP decapsulation
else
    if dot11PrivacyOptionImplemented is “true”
        if (the MPDU has individual RA and
        there is an entry in dot11WEPKeyMappings matching the MPDU’s TA)
            if that entry has WEPOn set to “false”
```

discard the frame body and increment *dot11WEPUndecryptableCount*
else
if that entry contains a key that is null
discard the frame body and increment *dot11WEPUndecryptableCount*
else
WEP decapsulate with that key, incrementing *dot11WEPICVErrorCount* if the ICV check fails
else
if *dot11WEPDefaultKeys*[KeyID] is null
discard the frame body and increment *dot11WEPUndecryptableCount*
else
WEP decapsulate with *dot11WEPDefaultKeys*[KeyID], incrementing *dot11WEPICVErrorCount* if the ICV check fails
else
discard the frame body and increment *dot11WEPUndecryptableCount*

8.2.3 Security association management

Pre-RSN security does not have a proper notion of a security association. Pre-RSN security possesses only one of the attributes, an authentication framework.

8.2.3.1 Authentication

8.2.3.1.1 Overview

The 1999 issue of the standard defines two subtypes of pre-RSN authentication service, *Open System* and *Shared Key*. Shared Key authentication is deprecated, and should not be implemented except for backward compatibility with legacy equipment. All management frames of subtype Authentication shall be unicast, as authentication is performed between pairs of stations—i.e., multicast authentication is not allowed. Management frames of subtype Deauthentication are advisory, and may be sent as group-addressed frames.

A mutual authentication relationship shall exist between two stations following a successful authentication exchange. Authentication shall be used between stations and the AP in an infrastructure BSS. Authentication may be used between two STAs in an IBSS.

8.2.3.1.2 Open system authentication

Open System authentication is a null authentication algorithm. Any STA requesting Open System authentication may be authenticated if dot11AuthenticationType at the recipient station is set to Open System authentication. A STA may decline to authenticate with another requesting STA. Open System authentication is the default authentication algorithm for pre-RSN equipment.

Open System authentication utilizes a two-message authentication transaction sequence. The first message asserts identity and requests authentication. The second message returns the authentication result. If the result is "successful," the STAs shall be declared mutually authenticated.

In the following description, the STA initiating the authentication exchange is referred to as the *requester*, and the STA to which the initial frame in the exchange is addressed is referred to as the *responder*.

8.2.3.1.2.1 Open System authentication (first frame)

— Message type: Management
— Message subtype: Authentication

— Information items:
• Authentication Algorithm Identification = “Open System”
• Station Identity Assertion (in SA field of header)
• Authentication transaction sequence number = 1
• Authentication algorithm dependent information (none)
— Direction of message: From requester to responder.

8.2.3.1.2.2 Open System authentication (final frame)

— Message type: Management
— Message subtype: Authentication
— Information items:
• Authentication Algorithm Identification = “Open System”
• Authentication transaction sequence number = 2
• Authentication algorithm dependent information (none)
• The result of the requested authentication as defined in 7.3.1.9
— Direction of message: From responder to requester.

If dot11AuthenticationType does not include the value “Open System,” the result code shall not take the value “successful.”

8.2.3.1.3 Shared key authentication

Shared Key authentication seeks to authenticate STAs as either a member of those who know a shared secret key or a member of those who do not. Shared Key authentication fails to meet this objective, as it makes public all the information required to trivially recover the key stream used by authentication.

Shared Key authentication requires the WEP privacy mechanism. Shared Key authentication shall be implemented if WEP is implemented.

This mechanism uses a shared key delivered to participating STAs via a secure channel that is independent of IEEE 802.11. This shared key is contained in a write-only MIB attribute in an attempt to keep the key value internal to the MAC.

A STA shall not initiate a Shared Key authentication exchange unless its dot11PrivacyOptionImplemented attribute is “true.”

In the following description, the STA initiating the authentication exchange is referred to as the *requester*, and the STA to which the initial frame in the exchange is addressed is referred to as the *responder*.

8.2.3.1.3.1 Shared Key authentication (first frame)

— Message type: Management
— Message subtype: Authentication
— Information Items:
• Station Identity Assertion (in SA field of header)
• Authentication Algorithm Identification = “Shared Key”
• Authentication transaction sequence number = 1
• Authentication algorithm dependent information (none)
— Direction of message: From requester to responder

8.2.3.1.3.2 Shared Key authentication (second frame)

Before sending the second frame in the Shared Key authentication sequence, the responder shall use WEP to generate a string of octets to be used as the authentication challenge text.

— Message type: Management

— Message subtype: Authentication
— Information Items:
- Authentication Algorithm Identification = “Shared Key”
- Authentication transaction sequence number = 2
- Authentication algorithm dependent information = the authentication result.
- The result of the requested authentication as defined in 7.3.1.9.

If the status code is not “successful,” this shall be the last frame of the transaction sequence, and the content of the challenge text field is unspecified.
If the status code is “successful,” the following additional information items shall have valid contents:

Authentication algorithm dependent information = challenge text.

This authentication result shall be of fixed length of 128 octets. The field shall be filled with octets generated by the WEP pseudo-random number generator (PRNG). The actual value of the challenge field is unimportant, but the value shall not be a static value.

— Direction of message: From responder to requester

8.2.3.1.3.3 Shared Key authentication (third frame)

The requester shall copy the challenge text from the second frame into the third frame. The third frame shall be transmitted after encapsulation by WEP, as defined in Clause 8.2.2.1, using the shared key.
— Message type: Management

— Message subtype: Authentication
— Information Items:
- Authentication Algorithm Identification = “Shared Key”
- Authentication transaction sequence number = 3
- Authentication algorithm dependent information = challenge text from the second frame

— Direction of message: From requester to responder

8.2.3.1.3.4 Shared Key authentication (final frame)

The responder shall WEP decapsulate the third frame as described in Clause 8.2.2.1. If the WEP ICV check is successful, the responder shall compare the decrypted contents of the Challenge Text field with the challenge text sent in second frame. If they are the same, then the responder shall respond with a successful status code in the final frame of the sequence. If the WEP ICV check fails or challenge text comparison fails, the responder shall respond with an unsuccessful status code in final frame.

— Message type: Management
— Message subtype: Authentication
— Information Items:
- Authentication Algorithm Identification = “Shared Key”
- Authentication transaction sequence number = 4
- Authentication algorithm dependent information = the authentication result

  The result of the requested authentication.
  This is a fixed length item with values “successful” and “unsuccessful.”

— Direction of message: From responder to requester

8.2.3.1.3.5 Shared key MIB attributes

To transmit a frame of type Management, subtype Authentication with an Authentication Transaction Sequence Number field value of 2, the MAC shall operate according to the following decision tree:

if *dot11PrivacyOptionImplemented* is “false” the MMPDU is transmitted with a sequence of zero octets in the Challenge Text field and a Status Code value of 13

else

the MMPDU is transmitted with a sequence of 128 octets generated using the WEP PRNG and a key whose value is unspecified and beyond the scope of this standard and a randomly chosen IV value (note that this will typically be selected by the same mechanism for choosing IV values for transmitted data MPDUs) in the Challenge Text field and a status code value of 0 (the IV used is immaterial and is not transmitted). Note that there are cryptographic issues involved in the choice of key/IV for this process as the challenge text is sent unencrypted and therefore provides a known output sequence from the PRNG.

To receive a frame of type Management, subtype Authentication with an Authentication Transaction Sequence Number field value of 2, the MAC shall operate according to the following decision tree:

if the Protected Frame subfield of the Frame Control field is 1

respond with a status code value of 15

else

if *dot11PrivacyOptionImplemented* is “true”

if there is a mapping in *dot11WEPKeyMappings* matching the MSDU’s TA

if that key is null

respond with a frame whose Authentication Transaction Sequence Number field is 3 that contains the appropriate Authentication Algorithm Number, a status code value of 15 and no Challenge Text field, without encrypting the contents of the frame

else

respond with a frame whose Authentication Transaction Sequence Number field is 3 that contains the appropriate Authentication algorithm Number, a status code value of 0 and the identical Challenge Text field, encrypted using that key, and setting the key ID subfield in the IV field to 0

else

if *dot11WEPDefaultKeys*[*dot11WEPDefaultKeyID*] is null

respond with a frame whose Authentication Transaction Sequence Number field is 3 that contains the appropriate Authentication Algorithm Number, a status code value of 15 and no Challenge Text field, without encrypting the contents of the frame

else

respond with a frame whose Authentication Transaction Sequence Number field is 3 that contains the appropriate Authentication Algorithm Number, a status code value of 0 and the identical Challenge Text field, WEP encapsulating the frame under the key *dot11WEPDefaultKeys*[*dot11WEPDefaultKeyID*], and setting the key ID subfield in the IV field to *dot11WEPDefaultKeyID*

else

respond with a frame whose Authentication Transaction Sequence Number field is 3 that contains the appropriate Authentication Algorithm Number, a status code value of 13 and no Challenge Text field, without encrypting the contents of the frame

When receiving a frame of type Management, subtype Authentication with an Authentication Transaction Sequence Number field value of 3, the MAC shall operate according to the following decision tree:

if the Protected Frame subfield of the Frame Control field is zero

```
        respond with a status code value of 15
else
        if dot11PrivacyOptionImplemented is "true"
                if there is a mapping in dot11WEPKeyMappings matching the MSDU's TA
                        if that key is null
                                respond with a frame whose Authentication Transaction Sequence Number field is 4 that contains the appropriate Authentication Algorithm Number, and a status code value of 15 without encrypting the contents of the frame
                        else
                                WEP decapsulate with that key, incrementing dot11WEPICVErrorCount and responding with a status code value of 15 if the ICV check fails
                else
                        if dot11WEPDefaultKeys[KeyID] is null
                                respond with a frame whose Authentication Transaction Sequence Number field is 4 that contains the appropriate Authentication Algorithm Number, and a status code value of 15 without encrypting the contents of the frame
                        else
                                WEP decapsulate with dot11WEPDefaultKeys[KeyID], incrementing dot11WEPICVErrorCount and responding with a status code value of 15 if the ICV check fails
        else
                respond with a frame whose Authentication Transaction Sequence Number field is 4 that contains the appropriate Authentication Algorithm Number, and a status code value of 15
```

The attribute *dot11PrivacyInvoked* shall not take the value "true" if the attribute *dot11PrivacyOptionImplemented* is "false." Setting *dot11WEPKeyMappings* to a value that includes more than *dot11WEPKeyMappingLength* entries is illegal and shall have an implementation-specific effect on the operation of the privacy service. Note that *dot11WEPKeyMappings* may contain from zero to *dot11WEPKeyMappingLength* entries, inclusive.

The values of the attributes in the aPrivacygrp should not be changed during the authentication sequence, as unintended operation may result.

8.3 RSN data privacy protocols

An RSN defines three data privacy protocols, named TKIP, WRAP, and CCMP. This section defines these protocols.

8.3.1 Overview

This standard defines three RSN data privacy protocols, TKIP, WRAP, and CCMP. TKIP provides pre-RSN hardware devices with a way to securely interoperate with RSN-capable devices. WRAP and CCMP are both protocol based on 128-bit AES, the first in OCB mode, and the second in CCM mode.

CCMP shall be mandatory-to-implement in all IEEE 802.11 equipment claiming RSN compliance. Implementation of TKIP and WRAP is optional for RSN compliance. Pre-RSN devices may be patched to implement TKIP, to interoperate with RSN-compliant devices that also implement TKIP. Use of any of the privacy algorithms depends on local policies.

Because of its weakness, IEEE 802.11 recommends not using TKIP except as a patch to pre-RSN equipment. RSN devices should implement TKIP only to allow interoperability with pre-RSN hardware implementing the TKIP patch.

8.3.2 Temporal Key Integrity Protocol (TKIP)

8.3.2.1 TKIP overview

The Temporal Key Integrity Protocol (TKIP) is a cipher suite enhancing the WEP protocol on pre-RSN hardware. This protocol uses WEP. TKIP surrounds WEP with new algorithms:

1. A transmitter calculates a keyed cryptographic *message integrity code*, or MIC, over the MSDU source and destination addresses and the MSDU plaintext data. TKIP appends the computed MIC to the MSDU data prior to fragmentation into MPDUs. The receiver verifies the MIC after decryption, ICV checking, and reassembly of the MPDUs into an MSDU, and discards any received MSDUs with invalid MICs. This defends against forgery attacks, and allows the MIC to be computed by software on the host.

2. Because an adversary can compromise the TKIP MIC with relatively few messages, TKIP also implements *countermeasures*, to rate limit key updates. The countermeasures bound the probability of a successful forgery and the amount of information an attacker can learn about a key.

3. TKIP uses a packet *TKIP sequence counter*, or *TSC*, to sequence the MPDUs it sends. The receiver drops MPDUs received out of order; i.e., not received with strictly increasing sequence numbers. This provides a weak form of replay protection. TKIP encodes the packet sequence counter as a WEP IV, to communicate the TSC value from the sender to the receiver.

4. TKIP uses a cryptographic mixing function to combine a *temporal key* and the TSC into the WEP seed, which includes the WEP IV. The receiver recovers the TSC from a received MPDU and utilizes the mixing function to compute the same WEP seed needed to correctly decrypt the MPDU. The key mixing function is designed to defeat weak-key attacks against the WEP key.

8.3.2.1.1 TKIP encapsulation

TKIP enhances the WEP encapsulation with several additional functions, as depicted in Figure 12 below.

1. TKIP computes the MIC over the MSDU source address, destination address, priority, and data, and appends the computed MIC to the MSDU; TKIP discards any MIC padding prior to appending the MIC.

2. TKIP fragments the MSDU into one or more MPDUs; TKIP assigns a monotonically incrementing TSC value to each MPDU it generates, taking care that all the MPDUs generated from the same MSDU use counter values from the same 16-bit counter space.

3. For each MPDU, TKIP uses the key mixing function to compute the WEP seed.

4. TKIP represents the WEP seed as a WEP IV and RC4 key, and passes these with each MPDU to WEP for encapsulation. WEP uses the WEP seed as a WEP default key, identified by a key id associated with the temporal key.

In the figure TTAK denotes the intermediate key produced by the phase 1 of the TKIP mixing function (see 8.3.2.4.3); TTAK is short-hand for “TKIP mixed Transmit Address and Key”.

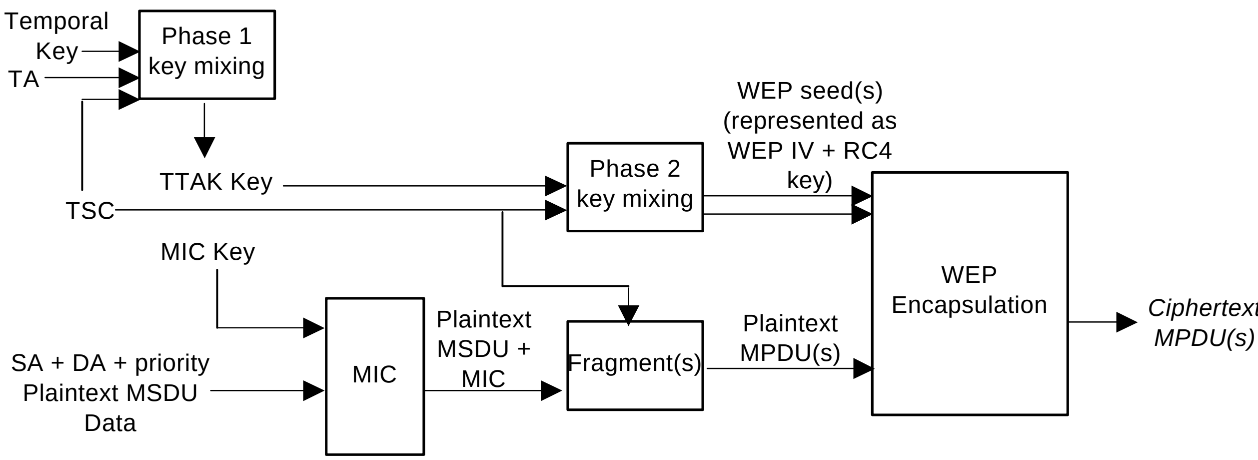


Figure 12—TKIP Encapsulation Block Diagram

8.3.2.1.2 TKIP decapsulation

TKIP enhances the WEP decapsulation process with the following additional steps.

1. Before WEP decapsulating a received MPDU, TKIP extracts the TSC sequence number and key id from the WEP IV. TKIP discards a received MPDU that violates the sequencing rules, and otherwise uses the mixing function to construct the WEP seed.

2. TKIP represents the WEP seed as a WEP IV and RC4 key and passes these with the MPDU to WEP for decapsulation.

3. If WEP indicates the ICV check succeeded, the implementation reassembles the MPDU into an MSDU. If the MSDU reassembly succeeds, the receiver verifies the MIC. If it fails, then the packet is discarded.

4. The MIC verification step recomputes the MIC over the MSDU source address, destination address, priority, and MSDU data (but not the MIC field), and bit-wise compares the result against the received MIC.

5. If the received and the locally computed MIC are identical, the verification succeeds, and TKIP shall deliver the MSDU to the upper layer. If the two differ in any bit position, then the verification fails, the receiver discards the packet, and engages in appropriate countermeasures.

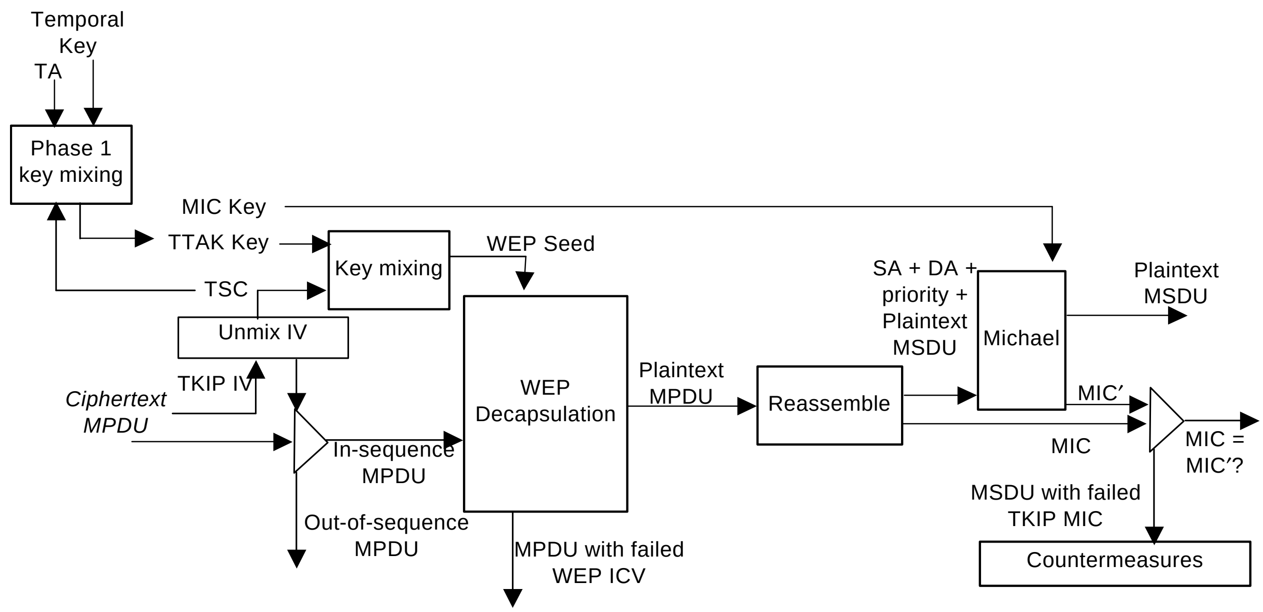


Figure 13—TKIP Decapsulation Block Diagram

8.3.2.2 TKIP MPDU formats

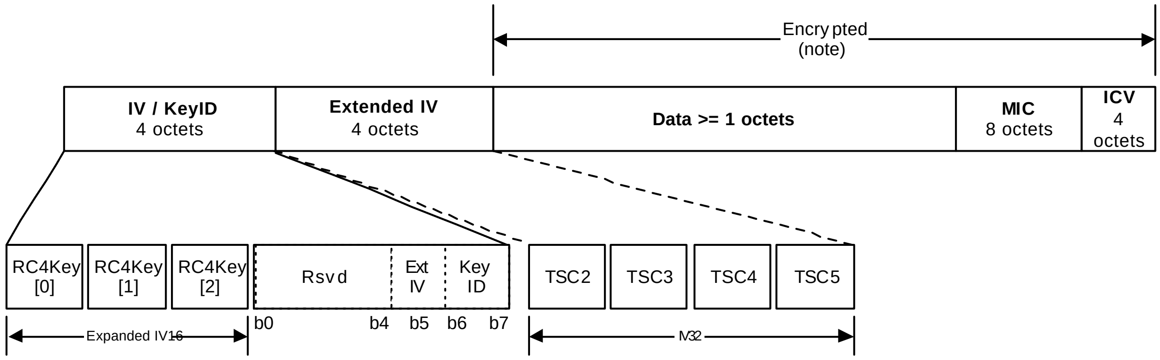
TKIP reuses pre-RSN WEP. It extends the MPDU by four (4) octets, to accommodate the an extension to the WEP IV, denoted by the Extended IV field, and extends the MSDU format by eight (8) octets, to accommodate the new MIC field. TKIP inserts the Extended IV field immediately after the WEP IV field and before the encrypted data. TKIP appends the MIC to the MSDU Data field; the MIC becomes part of the encrypted data.

Once the MIC is appended to the MSDU data, the TKIP data encapsulation can proceed in one of two ways.

- If the MSDU-with-MIC can be encoded within a single WEP-encapsulated MPDU, TKIP encapsulates the MSDU in a single MPDU.

- If the MSDU-with-MIC cannot be encoded within a single WEP-encapsulated MDPU, the MSDU-with-MIC is fragmented into appropriately sized MPDUs. WEP encapsulates each MPDU. Note that the MIC may span the second to last and last MPDUs.

Figure 14 below depicts the layout of the encrypted MPDU when using TKIP-based privacy. Note the Figure only depicts the case when the MSDU can be encapsulated



Note: The encipherment process has expanded the original MPDU size by 20 octets, 4 for the Initialization vector (IV) / Key ID field, 4 for the extended IV field, 8 for the Message Integrity Code (MIC) and 4 for the Integregty Check Value (ICV).

Figure 14—Construction of Expanded TKIP MPDU

The ExtIV bit in the KeyId octet indicates the presence or absence of an extended IV. If the ExtIV bit is ‘0’ only the old-style non-extended IV is transferred. If the ExtIV bit is ‘1’ an extended IV of 4 octets follows the original IV. For TKIP the ExtIV bit shall be set, and the Extended IV field shall be supplied. The ExtIV bit shall be 0 for WEP packets.

IV0 is the most significant octet of the IV and IV5 the least significant. Octets IV4 and IV5 form the IV sequence number part and are used with the TKIP phase 2 key hashing. Octets IV0 – IV3 are used in the TKIP phase 1 key hashing. It encodes the least significant 16 bits of the whole 48-bit IV. As soon as this lower 16 bit sequence number rolls over (0xFFFF $\rightarrow$ 0x0000), the extended IV value—i.e., the upper 32 bits of the entire 48-bit IV—must be incremented by 1.

Informational note: The rationale for this construction is:

- Aligning on word boundaries eases implementation on legacy devices
- Adding 4 octets of extended IV eliminates IV exhaustion as a reason to re-key.
- Retain IV/Key-ID of 4 octets, add 4 octets and use the last 2 octets (16bits) of the IV as the sequence number.
- Key ID octet changes – Use one bit (bit 5) to indicate that an extended IV is present. This allows the receiver/transmitter to know that the extended mode is present. The receiver/transmitter processes the following 4 octets as the extended IV. The receiving/transmitting station also uses the value of IV4 and IV5 octets to detect that a key rollover has occurred. When a key rollover has occurred, a new Phase 1 value is calculated, and used to decrypt the received/transmitted frame.

The extended IV field shall not be encrypted.

Note that if the TSC is represented as an octet string according to the conventions of 7.1.1, then

TSC = TSC0 TSC1 TSC2 TSC3 TSC4 TSC5

where TSC0 is the least significant octet and TSC5 the most significant. The mixing function uses the least significant octet of the TSC as RC4Key[0], and the second least significant octet at RC4[2]:

RC4Key[0] = TSC0 and RC4Key[2] = TSC1.

The effect of this construction is the TSC is encoded as a little-Endian integer in each TKIP MPDU. TKIP shall encrypt all the MPDUs generated from one MSDU under the same key.

8.3.2.3 TKIP state

TKIP augments the *dot11WEPKeyMappings* and dot11*WEPDefaultKeyTable* MIB arrays with two new variables each, respectively *dot11KeyMappingValue* and *dot11KeyMappingSize*, and *dot11DefaultKeyValue* and *dot11DefaultKeySize*. The variables dot11DefaultKeySize and *dot11KeyMappingSize* are integers and indicate the length of the key in octets in the *dot11DefaultKeyValue* and *dot11KeyMappingValue* variables, respectively. The variables *dot11DefaultKeyValue* and *dot11KeyMappingValue* are 32 octet strings in size and supply the TKIP encryption key, concatenated with the TKIP send and receive integrity keys, as described in Annex D.

8.3.2.4 TKIP procedures

8.3.2.4.1 TKIP MIC

Flaws in the original IEEE 802.11 WEP design caused it to fail to meet its goal of protecting data traffic content from casual eavesdroppers. Among the most significant flaws was it lack of a mechanism to defeat message forgeries and other active attacks. To defend against active attacks, TKIP requires a MIC, named *Michael*. Michael offers only weak defenses against message forgeries, but it constitutes the best that can be achieved with the majority of legacy hardware.

Annex F contains a “C++” language reference implementation of the TKIP MIC. It also provides test vectors for the MIC.

Informative Note: Before defining the details of the Michael MIC, it is useful to review the context in which this mechanism must work. Active attacks enabled by the original WEP design include:

- Bit-flipping attacks;
- Data (payload) truncation and concatenation;
- Fragmentation attacks;
- Iterative guessing attacks against the key;
- Redirection by modifying the MPDU DA or SA fields;
- Impersonation attacks by modifying the MPDU SA or TA fields.

The MIC makes it more difficult for any of these attacks to succeed.

With the Michael design, all of these attacks remain at the MPDU level. The MIC, however, applies to the MSDU, so blocks successful MPDU level attacks. TKIP applies the MIC to the MSDU at the transmitter and verifies it at the MSDU level at the receiver. If an MIC check fails at the MSDU level, the implementation shall discard the MSDU and invoke counter-measures.

Figure 15 depicts different peer layers communicating:

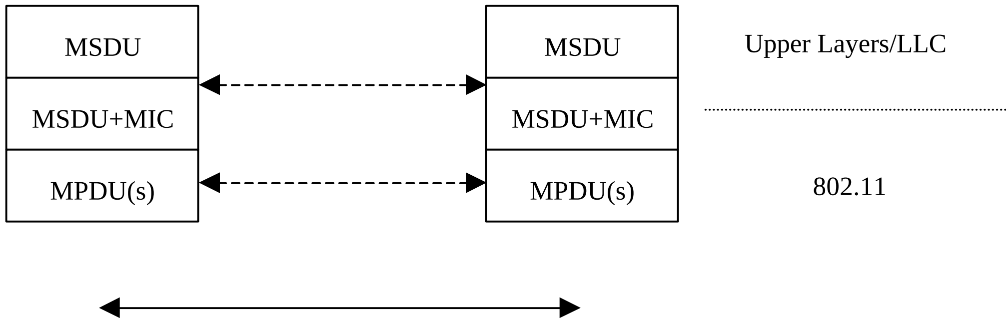


Figure 15—TKIP MIC Relation to 802.11 Processing (Informative)

The figure depicts an architecture whereby the MIC is logically appended to the raw MSDU in response to the MA-UNITDATA.request primitive. That is, the TKIP MIC is computed over

- the MSDU destination address (DA);
- the MSDU source address (SA);
- the MSDU priority; and
- the entire unencrypted MSDU data (payload).

DA	SA	Priority	0	Data	MIC

Note the DA, SA and a one octet priority field and 3 octet reserved (0) field are used for calculating the MIC and are not transmitted. The priority field shall be 0 and reserved for future use for IEEE 802.11 traffic class.

TKIP appends the MIC at the end of the MSDU payload, reducing the maximum allowed MSDU payload size by the size of the MIC field, which is 8 bytes for Michael. The IEEE 802.11 MAC then applies its normal processing to transmit this MSDU-with-MIC as a sequence of one or more MPDUs. This means the MSDU plus MIC can be partitioned into one or more MPDUs, the WEP ICV is calculated over each MDPU, and MIC can be partitioned across the final two MPDUs. The TKIP MIC augments but does not replace the WEP ICV. TKIP protects the MIC with encryption, because it is a weak construction; the encryption then makes MIC forgeries somewhat more difficult. The WEP ICV helps prevent false positives, whereby normal operation rather than attack corrupt the transmitted MIC value.

The receiver reverses this procedure to reassemble the MSDU, and, after the MSDU has been logically reassembled, the MAC verifies the MIC prior to delivery of the MSDU to upper layers. If the MIC validation succeeds, the MAC delivers the MSDU to the appropriate IEEE 802 SAP via the MA-UNITDATA.indication primitive. If the MIC validation fails, the MAC discards the MSDU, increments a counter, and invokes counter-measures.

TKIP calculates the MIC over the MSDU rather than the MPDU for two reasons. First, it detects attacks against MPDUs more easily than can be done at the MPDU level alone. Second, it increases the implementation flexibility, allowing the MIC to be implemented either within the STA hardware or in a software driver running on either the STA or the STA's host.

It should be noted that a MIC cannot provide complete forgery protection, as it cannot defend against replay attacks. TKIP provides replay detection by IV sequencing, ICV validation, and rekeying. Furthermore, if TKIP is utilized with a group key, an "insider" STA can masquerade as any other STA belonging to the group. Hence, the protection afforded by the TKIP MIC is directly affected by the local keying policy; group keys should be avoided.

Michael generates a 64-bit MIC, with a design goal of 20 bits of security. The Michael key consists of 64-bits, represented as an 8-byte sequence $k_0...k_7$. This is converted to two 32-bit little-Endian words K_0 and K_1. Throughout the Michael design, all conversions between bytes and 32-bit words shall use the little-Endian conventions, given in 7.1.1.

Michael operates on MSDUs. An MSDU consists of octets $m_0...m_{n-1}$ where n is the number of MSDU octets, including source address, destination address, and data field. The Michael algorithm does not interpret the MSDU data field, which typically begins with an IEEE 802 SNAP header. The message is padded at the end with a single byte with value 0x5a, followed by between 4 and 7 zero bytes. The number of zero bytes is chosen so that the overall length of the padded MSDU is a multiple of 4. The padding is not transmitted with the MSDU; it is used to simplify the computation over the final block. The MSDU is then converted to a sequence of 32-bit words $M_0 ...M_{N-1}$, where $N = \lceil (n+5)/4 \rceil$, and where $\lceil a \rceil$ means to round a up to the nearest integer. By construction $M_{N-1} = 0$ and $M_{N-2} \neq 0$.

The MIC value is computed iteratively by starting with the key value and applying a block function b for every message word, as shown in Figure 16. The algorithm loop runs a total of N times (i takes on the values 0 to N–1 inclusive), where N is as above, the number of 32-bit words comprising the padded MSDU. The algorithm results in two words (l,r), which are converted to a sequence of eight octets using the least-significant-octet-first convention. This is the MIC value. The MIC value is appended to the MSDU as data to be sent. Note that the padding is used in the MIC computation only, and is discarded prior to appending the MIC to the MSDU.

Input: Key (K_0, K_1) and padded MPDU (represented as32-bit words) $M_0...M_N$
Output: MIC value (V_0, V_1)
MICHAEL((K_0, K_1) , (M_0,...,M_N))
    $(l,r) \leftarrow (K_0, K_1)$
    for $i = 0$ **to** N–1 **do**
        $l \leftarrow l \oplus M_i$
        $(l, r) \leftarrow b(l, r)$
    return (l,r)

Figure 16—Michael message processing

Figure 17 defines the Michael block function b. It is a Feistel-type construction with alternating additions and XOR operations. It uses <<< to denote the rotate-left operator on 32-bit values, >>> for the rotate-right operator, and XSWAP for a function that swaps the position of the two least significant bytes and the position of the two most significant bytes in a word.

Input: (l,r)
Output: (l,r)
*b(*L,R)
    $r \leftarrow r \oplus (l <<< 17)$
    $l \leftarrow (l + r) \bmod 2^{32}$
    $r \leftarrow r \oplus \mathrm{XSWAP}(l)$
    $l \leftarrow (l + r) \bmod 2^{32}$
    $r \leftarrow r \oplus (l <<< 3)$
    $l \leftarrow (l + r) \bmod 2^{32}$
    $r \leftarrow r \oplus (l >>> 2)$
    $l \leftarrow (l + r) \bmod 2^{32}$
    return (l, r)

Figure 17—Michael block function

8.3.2.4.2 TKIP counter-measures

Michael's design trades off security in favor of implementability on pre-RSN equipment. Michael provides only weak protection against active attack. A failure of the MIC in a received MSDU indicates a probable active attack. If TKIP implementation detects a probable active attack, TKIP shall take countermeasures as specified in this clause. These counter-measures accomplish the following goals:

- The current authentication key and encryption key shall be deleted and not used again. This prevents the attacker from learning anything about those keys from the MIC failure.
- Significant effort should be made to log the event as a security-relevant matter. A MIC failure is an almost certain indication of an active attack, and warrants a follow-up by the system administrator.
- The rate of MIC failures *must* be kept below one per minute. This implies that new keys must not be generated if devices frequently receive packets with forged MICs. The slowdown makes it difficult for an attacker to make a large number of forgery attempts in a short time.

Before verifying the MIC, the receiver shall check the CRC, ICV, and IV for all related MPDUs. MPDUs with invalid CRCs, ICVs, or with whose MPDUs' IVs falling before the IV window shall be discarded before checking the MIC. This avoids unnecessary MIC failure events. Checking the IV before the MIC makes countermeasure-based DOS attacks harder to perform.

If an Authenticator's STA detects a MIC failure on a received TKIP-protected MSDU, it shall take the following steps:

1. For an MSDU which was protected with a Group key:
    a. Delete the Group encryption and integrity keys in question.
    b. Wait until 60 seconds have occurred from the last MIC failure (either from an EAPOL-Key message with a MIC failure or a local MIC failure occurred).
    c. Update the Group Transient Key to all associated stations.
    d. Log details of the MIC failure.
2. For an MSDU which was protected with a Pairwise Key:
    a. Drop any received data messages except IEEE 802.1X messages until the Pairwise Key is deleted or changed.
    b. Wait until 60 seconds have occurred from the last MIC failure (either from an EAPOL-Key message with a MIC failure or a local MIC failure).
    c. Initiate a 4-way handshake with the peer STA to reestablish a new Pairwise key.
    d. Log details of the MIC failure.

An AP shall drop any data broadcast/multicast MSDU received from a non-AP STA.

If a Supplicant's STA detects a MIC failure, it shall take the following steps:

1. For an MSDU which was encrypted with a Group Key:
    a. Delete the Group encryption and integrity keys in question.

b. Send an EAPOL-Key message requesting for a new Group key.

c. Log details of the MIC failure at the station and AP.

2. For an MSDU which was protected with a Pairwise Key:

a. Drop any received data messages except IEEE 802.1X messages until the Pairwise Key is deleted or changed.

b. Send an EAPOL-Key message requesting for a new Pairwise key.

c. Log details of the MIC failure at the peer STAs.

An EAPOL-Key message from Supplicant to Authenticator with Request bit set asks the Authenticator to change the indicated key.

After Michael failure detected either locally or is signaled by a received EAPOL-Key Request, the Authenticator shall generate and distribute at most one replacement key during the 60 seconds following the error. This means that when a Michael failure occurs involving a Group key, the Authenticator generates and distributes a new GTK to all associated stations if a second Michael failure involving the Group Key has not been detected within the prior 60 seconds. If a second failure occurs within the 60 second window, the Authenticator waits a full 60 seconds before generating and distributing another replacement key. Similarly, if a Michael failure involving a Pairwise Key occurs, the Authenticator shall generate and distribute a replacement PTK via a 4-way handshake if it detects no other Michael failure involving a PTK within 60 seconds of the Michael failure. If a second failure is detected within 60 seconds of a previous Michael failure, the Authenticator shall wait a full 60 seconds before replacing the PTK.

Note that Michael failures delay the generation and distribution keys to STAs other than those involved in the failure. This prevents an attacker attacking a Michael key, then forcing the STA to re-associate, and then repeating the attack cycle.

8.3.2.4.3 TKIP mixing function

Annex F defines the TKIP S-box, a “C” language reference implementation of the TKIP mixing function. It also provides test vectors for the mixing function.

The mixing function has two phases. The first phase mixes the dot11DefaultKeyValue or dot11KeyMappingValue (TK) with the transmitter address (TA) and TSC. A STA may cache the output of this phase to reuse with subsequent MPDUs associated with the same TK and TA. The second phase mixes the output of the first phase with the TSC and TK to produce the WEP seed, also called the per-packet key. The WEP seed may be computed well before it is used. The two-phase process may be summarized as:

TTAK ← Phase1(TK, TA, TSC)
WEP seed ← Phase2(TTAK, TSC)

Phase 1 is somewhat simpler than Phase 2. This simplicity is possible because the output of Phase 1 is not used directly as an RC4 key.

Both Phase 1 and Phase 2 rely on an S-box, defined in Annex F. The S-box substitutes one 16-bit value with another 16-bit value. This function is a non-linear substitution, and may be implemented as a table look up.

Phase 1 Definition. The inputs to the first phase of the temporal key mixing function shall be a dot11DefaultKeyValue or dott11KeyMappingValue (*TK*), the transmitter address (*TA*), and the TSC. The *TK* shall be 128 bits in length. Only the most significant 32 bits of the TSC and the first 80 bits of TK are used in Phase 1. The output, called *TTAK*, shall be 80 bits in length and is represented by an array of 16-bit values $TTAK_0$ $TTAK_1$ $TTAK_2$ $TTAK_3$ $TTAK_4$.

The description of the phase 1 algorithm treats all of the following values as arrays of 8-bit values: $TA_0..TA_5$, $TK_0..TK_{12}$. The *TA* byte order is represented according to the conventions from 7.1.1, and the first three bytes represent the OUI.

The exclusive-or (⊕) operation, the bit-wise-and (&) operation, and the addition (+) operation are used in the Phase 1 specification. . A loop counter, called *i*, and an array index temporary variable, called *j*, are also employed.

One function, *Mk*16, is used in the definition of Phase 1. The function Mk16 constructs a 16-bit value from two 8-bit inputs as $Mk16(X,Y) = 256 \cdot X+Y$.

Two steps comprise the phase 1 algorithm. The first step initializes *TTAK* from *TSC* and *TA*. The second step uses an S-box to iteratively mix the keying material into the 80-bit TTAK. The second step sets the PHASE1_LOOP_COUNT to 8.

Input: transmit address $TA_0 \ldots TA_5$, temporal key $TK_0..TK_{12}$, and $TSC_0..TSC_2$
Output: intermediate key $TTAK_0..TTAK_4$

```
PHASE1-KEY-MIXING(TA_0…TA_5, TK_0..TK_12, TSC_0..TSC_2)
        PHASE1_STEP1:
        TTAK_0 ← TSC_0
        TTAK_1 ← TSC_1
        TTAK_2 ← Mk16(TA_1,TA_0)
        TTAK_3 ← Mk16(TA_3,TA_2)
        TTAK_4 ← Mk16(TA_5,TA_4)

        PHASE1_STEP2:
        for i = 0 to PHASE1_LOOP_COUNT-1
                j ← 2·(i & 1)
                TTAK_0 ← TTAK_0 + S[TTAK_4 ⊕ Mk16(TK_1+j,TK_0+j)]
                TTAK_1 ← TTAK_1 + S[TTAK_0 ⊕ Mk16(TK_5+j,TK_4+j)]
                TTAK_2 ← TTAK_2 + S[TTAK_1 ⊕ Mk16(TK_9+j,TK_8+j)]
                TTAK_3 ← TTAK_3 + S[TTAK_2 ⊕ Mk16(TK_13+j,TK_12+j)]
                TTAK_4 ← TTAK_4 + S[TTAK_3 ⊕ Mk16(TK_1+j,TK_0+j)] + i
        end
```

Figure 18—Phase 1 key mixing

Phase 2 Definition. The inputs to the second phase of the temporal key mixing function shall be the output of the first phase (*TTAK*) together with the TK and the TKIP sequence counter *TSC*. The *TTAK* is 80-bits in length. The *TSC* is 48 bits. Only the last 24 bits of TK are used in Phase 2. The output is the WEP seed, which is a per-packet key, and is 128-bits in length. The constructed WEP seed has an internal structure conforming to the WEP specification. That is, the first 24 bits of the WEP seed shall be transmitted in plaintext as the WEP IV. As such, these 24 bits are used to convey lower 16 bits of the TSC from the sender (encryptor) to the receiver (decryptor). The rest of the TSC shall be conveyed in the EIV field, in big-Endian order. The TK and TTAK values are represented as in Phase 1. The WEP seed is treated as an array of 8-bit values: $Seed_0 \ldots Seed_{15}$. The *TSC* shall be treated as an array of 16-bit value TSC_0 TSC_1 TSC_2.

The pseudo code specifying the Phase 2 mixing function employs one variable: PPK. PPK is 128-bits, and it is represented as an array of 16-bit values: $PPK_0..PPK_7$. The pseudo code also employs a loop counter, called *i*. As detailed below, the mapping from the 16-bit *PPK* values to the 8-bit *WEPseed* values is explicitly little-Endian to match the Endian architecture of the most common processors used for this application.

The exclusive-or operation (⊕), the addition operation (+), the and operation (&), the or operation (|), and the right bit shift operation (>>) are used the specification of Phase 2 below.

The algorithm specification relies on four functions.

- The first function, *Lo*8, references the least significant 8 bits of the 16-bit input value.
- The second function, *Hi*8, references the most significant 8 bits of the 16-bit value.
- The third function *RotR*1 rotates its 16-bit argument 1 bit to the right.
- The fourth function is *Mk*16, already used in Phase 1, defined by $Mk16(X,Y) = 256 \cdot X+Y$, and constructs a 16-bit output from two 8 bit inputs.

  Note: The rotate and addition operations in STEP2 makes Phase 2 particularly sensitive to the Endian architecture of the processor, although the performance degradation due to running this algorithm on a big-Endian processor should be minor.

The second phase is comprised of three steps.

- STEP1 makes a copy of the TTAK and brings in the TSC.
- STEP2 is a 96-bit bijective mixing, employing an S-box.
- STEP3 brings in the last of the TK bits and assigns the 24-bit WEP IV value.

Input: intermediate key $TTAK_0 \ldots TTAK_4$, *TK*, and TKIP sequence counter *TSC*
Output: WEP Seed $WEPSeed_0 \ldots WEPSeed_{15}$

```
PHASE2-KEY-MIXING(TTAK_0…TTAK_4, TK, TSC)
    PHASE2_STEP1:
    PPK_0 ← TTAK_0
    PPK_1 ← TTAK_1
    PPK_2 ← TTAK_2
    PPK_3 ← TTAK_3
    PPK_4 ← TTAK_4
    PPK_5 ← TTAK_4 + TSC

    PHASE2_STEP2:
    PPK_0 ← PPK_0 + S[PPK_5 ⊕ Mk16(TK_1,TK_0)]
    PPK_1 ← PPK_1 + S[PPK_0 ⊕ Mk16(TK_3,TK_2)]
    PPK_2 ← PPK_2 + S[PPK_1 ⊕ Mk16(TK_5,TK_4)]
    PPK_3 ← PPK_3 + S[PPK_2 ⊕ Mk16(TK_7,TK_6)]
    PPK_4 ← PPK_4 + S[PPK_3 ⊕ Mk16(TK_9,TK_8)]
    PPK_5 ← PPK_5 + S[PPK_4 ⊕ Mk16(TK_11,TK_10)]
    PPK_0 ← PPK_0 + RotR1(PPK_5 ⊕ Mk16(TK_13,TK_12))
    PPK_1 ← PPK_1 + RotR1(PPK_0 ⊕ Mk16(TK_15,TK_14))
    PPK_2 ← PPK_2 + RotR1(PPK_1)
    PPK_3 ← PPK_3 + RotR1(PPK_2)
    PPK_4 ← PPK_4 + RotR1(PPK_3)
    PPK_5 ← PPK_5 + RotR1(PPK_4)

    PHASE2_STEP3:
    WEPSeed_0 ←  Hi8(TSC)
    WEPSeed_1 ← (Hi8(TSC) | 0x20) & 0x7F
    WEPSeed_2 ←  Lo8(TSC)
    WEPSeed_3 ←  Lo8((PPK_5 ⊕ Mk16(TK_1,TK_0)) >> 1)
    for i = 0 to 5
```

```
            WEPSeed_{4+(2·i)} ← Lo8(PPK_i)
            WEPSeed_{5+(2·i)} ← Hi8(PPK_i)
        end
        return WEPSeed_0…WEPSeed_15
```

Figure 19—Phase 2 key mixing

The WEP IV format carries three octets. Step 3 of Phase 2 determines the value of each of these three octets. The construction was selected to preclude the use of known weak keys. The recipient can reconstruct the least significant 16 bits of the TSC used by the originator by concatenating the first and third octets, ignoring the second octet. The remaining 32 bits of the TSC are obtained from the EIV.

Informative Note: S-box. The algorithm S-box utilized by the Phase 1 and Phase 2 functions is defined in Annex F. The S-box substitutes one 16-bit value with another 16-bit value. This is a non-linear substitution. The reference implementation in Annex F implements as a table look-up. The table look-up can be organized as either a single table with 65,536 entries and a 16-bit index (128 Kbytes of table) or two tables with 256 entries and an 8-bit index (1024 bytes for both tables). When the two smaller tables are used, the high-order byte is used to obtain a 16-bit value from one table and the low-order byte is used to obtain a 16-bit value from the other table; the S-box output is the exclusive-or (⊕) of the two 16-bit values. The second S-box table is a byte-swapped replica of the first.

The sample code in Annex F uses the two smaller table approach. The S-box tables can be extracted from the AES reference implementation.

Informative Note: The transmitter address (TA) is mixed into the temporal key (TK) in the first phase of the hash function. Implementations can achieve a significant performance improvement by caching the output of the first phase. The Phase 1 output is the same for 2^{16} = 65,536 consecutive packets from the same TK and TA. Consider the simple case where a station communicates only with an access point (AP). The station will perform the first phase using its own address, and it will be used to encrypt traffic sent to the access point. The station will perform the first phase using the access point address, and it will be used to decrypt traffic received from the access point.

With TSC 48 bits in size the key caches will need to be updated when the lower 16 bits of the TSC wrap and the upper 32 bits need to be updated.

8.3.2.4.4 TKIP replay protection

TKIP implementations shall reuse the WEP IV field to defend against replay attacks by implementing the following rules.

1. As with WEP IVs, TKIP TSC values shall correspond to MPDUs.

2. The TSC (48 bit counters) shall be selected from a single pool by each transmitter for each temporal key—i.e., each transmitter has its own unique counter for each directional temporal key established.

3. The TSC shall be implemented as a 48-bit monotonically incrementing counter, *initializ*ed to zero when the corresponding TKIP temporal key is *initializ*ed or refreshed.

4. The WEP IV format carries the least significant 16 bits of the 48-bit TSC, as defined by the TKIP mixing function phase 2 step 3. The remainder of the TSC is carried in the EIV.

5. A receiver shall maintain a separate set of TKIP replay windows for each MAC address it receives TKIP traffic from. The receiver initializes the replay window whenever it resets the temporal key for a peer.

Informative Note: The per-MAC address condition in 5 is needed to accommodate multicast/broadcast keys in the IBSS case.

6. A receiver shall delay advancing a TKIP replay window until an MSDU passes the MIC check, to prevent attackers from injecting MPDUs with valid ICVs and IVs but invalid MICs.

7. In order to accommodate burst ACK, the TKIP receiver shall check that the received TSC (48 bit counter) is no smaller than 15 less than the greatest TKIP replay window value for the MPDU's temporal key. When combined with the prohibition on correctly decrypting more than one MPDU under a given <temporal key, IV> pair, this provides replay protection and accommodates frames that may be delayed due to message class priority values, with a window size of 16.

Note: This works because if an attacker modifies the IV, then this alters the encryption key and hence both the ICV and MIC will ordinarily decrypt incorrectly, causing the received MPDU to be dropped.

8.3.3 Wireless Robust Authenticated Protocol (WRAP)

A cipher suite based on the ***Advanced Encryption Standard*** (AES) and ***Offset Codebook*** (OCB) mode has been adopted. This cipher suite is called *Wireless Robust Authenticated Protocol* (*WRAP*) ***privacy***, and this clause defines it. Support for this protocol is optional.

8.3.3.1 WRAP overview

WRAP privacy consists of three parts: a key derivation procedure, an encapsulation procedure, and a decapsulation procedure. It is based on 128-bit AES in OCB mode.

a) The encapsulation procedure. Once the key has been derived and its associated state *initializ*ed, the IEEE 802.11 MAC uses the WRAP encapsulation algorithm with the key and the state to protect all unicast MSDUs it sends to an associated station.

b) The decapsulation procedure. Similarly, once the key has been derived and associated state *initializ*ed, the IEEE 802.11 MAC uses the WRAP decapsulation algorithm with the receive key and state to decapsulate all unicast MSDUs received from an associated station. Once the key is established, the MAC shall discard any MSDUs received over the association that are unprotected by the encapsulation algorithm.

IEEE 802.1X may also assign a broadcast/multicast key. The implementation uses this key as configured, without derivation. The MAC utilizes the broadcast/multicast key to protect all broadcast/multicast MSDUs it sends, and discards any broadcast/multicast MSDUs received that are not protected by this key.

Informative Note 1. The WRAP privacy protocol requires IEEE 802.1X authentication and key management.

Informative Note 2.The quality of protection any key offers with any cryptographic algorithm degrades through key usage. It is impossible to estimate when the protection a key affords has been exhausted without counting the number of blocks protected. In order to avoid maintaining a history of all MSDUs used with every key, this means that a fresh, never-used-before key is required whenever a new "session" begins, so that keys cannot be used independently of some notion of a session. Similarly, the replay protection counter requires that peers synchronize a fresh key whenever they reinitialize the replay state.

Informative Note 3. The WRAP privacy protocol architecturally lies above the IEEE 802.11 retry function. This is required since an MSDU may be accepted by the local IEEE 802.11 implementation but its acknowledgement lost in transit to the peer. If the WRAP privacy protocol were to lie below the IEEE 802.11 MAC retry function, then it would be impossible to recover from this state, as the replay protection function would discard all further retries.

AES is defined by FIPS Standard 197. Annex G defines OCB Mode.

8.3.3.1.1 WRAP encapsulation

The following steps encapsulate MSDU plaintext data:

a) Select the appropriate context based on the MSDU;

b) Increment block count and the appropriate replay counter, based on the MSDU service class;

c) Construct the Replay-Counter field of the final WRAP-protected MSDU payload;

d) Construct the OCB nonce using the Replay-Counter, MSDU service class, and source MAC address;

e) Construct an associated data block from the destination MAC address;

f) AES-OCB encrypt the MSDU and associated data;

g) Construct the MSDU payload from the replay counter, OCB encrypted data, and the OCB tag.

8.3.3.1.2 WRAP decapsulation

The following steps decapsulate data an MSDU received over a protected association or broadcast/multicast channel:

a) Select the appropriate context based on the received MSDU;

b) perform some basic sanity checks on the packet (See 8.3.3.4.8);

c) construct the OCB nonce using the Replay-Counter, QoS Traffic Class, and the source and destination MAC addresses from the received MSDU;

d) using the constructed nonce and temporal key from the selected context, WRAP decrypt the MSDU data;

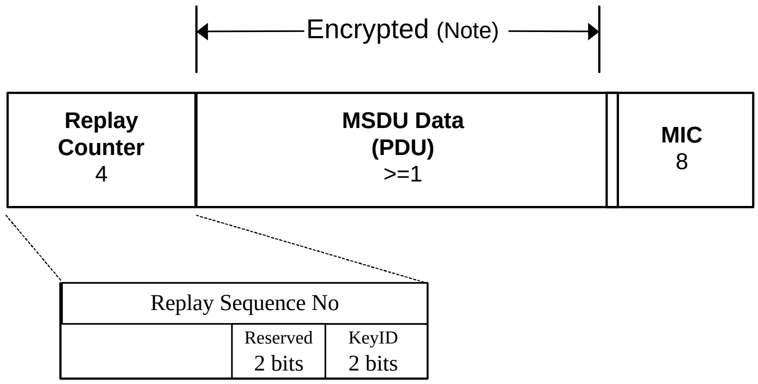
e) If the MSDU is unicast, extract the sequence number from the MSDU Replay-Counter field and verify the MSDU is not a replay.

Note. It is infeasible to provide replay protection for multicast/broadcast MSDUs using symmetric key techniques, and asymmetric key techniques are too computationally expensive to employ for datagram traffic.

8.3.3.2 WRAP MSDU format

The WRAP privacy method encapsulates the MSDU payload. Figure 20 shows the encapsulated MSDU when using WRAP privacy.

The data overhead of the WRAP privacy algorithm is 12 octets. This includes a 28-bit replay counter, the single KeyID octet inherited from WEP, and a 64-bit Message Integrity Code (MIC) used to detect forgeries.



Note: The encipherment process has expanded the original MSDU by 12 Octets, 4 for the replay counter field, and 8 for the Message Integrity Check (MIC). The MIC is calculated over the Data fields only.

Figure 20 - Construction of Expanded WRAP MSDU

The WRAP privacy protocol is invisible to entities outside the IEEE 802.11 MAC data path.

Note: The AES-OCB-protected MSDU payload may span MPDUs.

8.3.3.3 WRAP state

WRAP privacy uses a MIB array called the *dot11WrapKeyMappings*. This support zero, one, or two entries for each MAC address pair with which the STA maintains secure associations. The size of the *dot11WrapKeyMappings* array is implementation-specific. A global MIB variable *dot11WrapKeyMappingLength* indicates the number of entries in the array.

Each entry of the *dot11WrapKeyMappings* groups together the following state:

1. A *dot11WrapReceiveAddress* and a *dot11WrapTransmitAddress*, indicating that this entry applies to all MSDUs being sent between this pair of addresses;

2. A *dot11WrapKeyID*, indicating the WEP KeyID into which this entry maps;

3. A 128-bit key called the *dot11AESOCBTemporalKey*, referred to informally as the temporal key. This is the derived key as specified in 8.3.1.3.4.1 for unicast, and the unaltered temporal key for broadcast/multicast. Both keys shall be configured by IEEE 802.1X.

4. A set of 28-bit counters called the *dot11WrapTrafficClassNSequenceCounter*, for constructing the next OCB nonce. *N* ranges from 0 to15, with one traffic class defined for each QoS service class. When QoS is not used, only *dot11WrapTrafficClass0SequenceCounter* is used.

5. A 48-bit counter *dot11WrapBlocksSent*, counting the number of 128-bit blocks protected by the present temporal key;

6. A set of 28-bit replay windows called the *dott11WrapTrafficClassNReplayWindow*, for detecting replays. *N* ranges from 0 to15. When QoS is not used, only *dot11WrapTrafficClasse0ReplayWindow* is used.

7. A boolean flag called *dot11WrapEnableTransmit*, to indicate when the temporal key and MIC send key can be used for transmitting MSDUs;

8. A boolean flag called *dot11WrapEnableReceive*, to indicate when the temporal key and MIC receive key can be used for receiving MSDUs.

9. a 32-bit counter *dot11WrapFormatErrors*, to indicate the number of MSDUs received with an invalid format, *initializ*ed to zero;

10. a 32-bit counter *dot11WrapReplays*, to indicate the number of received unicast fragments discarded by the replay mechanism, *initializ*ed to zero;

11. a 32-bit counter *dot11WrapDecryptErrors*, to indicate the number of received fragments discarded by the OCB decryption mechanism, *initializ*ed to zero; and

12. a 48-bit counter *dot11WrapRecvdBlocks*, to track the total number of protected blocks received.

Informative Note 1: A broadcast/multicast entry does not utilize the replay window. This is because it is impossible to detect broadcast/multicast replays using symmetric key techniques. In particular, any party holding the broadcast/multicast key can masquerade as any other member of the group, so can intrude on another's sequence space without detection.

Informative Note 2: As an optimization, implementations may compute and maintain the AES-OCB key schedule rather than maintain the temporal key.

8.3.3.4 WRAP procedures

8.3.3.4.1 Transmit context selection

To encapsulate data, the transmitter first checks whether the MSDU is unicast or multicast/broadcast. It selects the correct transmit context by mapping the destination address to an entry in the *dot11WrapKeyMappings*. If an appropriate context exists, a conformant implementation shall use the entry to protect any MSDU it sends.

8.3.3.4.2 Incrementing the transmit block count and replay counter

To encapsulate data, the transmitter computes the total number of blocks to be protected in the MSDU. This is defined as

$$m = \lceil (\text{\# MSDU data octets})/\text{AES-Block-Size} \rceil,$$

where $\lceil a \rceil$ means, as before, to round a up to the nearest integer, and AES-Block-Size = 16 (octets).

If adding the number of blocks m would cause the context's value of *dot11WrapBlocksSent* to wrap—*i.e.*, if m + *dot11WrapBlocksSent* > 2^{48}—then the cryptographic protection afforded by the key are considered exhausted, and it is a protocol error to use the key any further. In this case, the encapsulation algorithm shall discard all transmit datagrams until the key is replaced with a new one.

Otherwise, from the selected context and the MSDU QoS traffic class, the implementation selects appropriate 28-bit per-service-class replay counter. If QoS traffic classes are not in use, there is only one replay counter for the entire association.

If the value of the selected replay counter is $2^{28}–2$ = 268435454 (or greater), then another valid nonce cannot be constructed. That is, reusing this replay counter means that more than one MSDU would be protected by the same <key, nonce> pair, voiding the security guarantees. Once again, the sender shall not transmit another MSDU on this association or broadcast/multicast channel until the key is replaced, and the encapsulation algorithm shall discard all datagrams until the key is replaced by a new one.

Otherwise, the value of the selected replay counter is less than 268435454, and it is still feasible to construct another valid nonce. The implementation adds *m* to *dot11AESOCBBlocksSent* and 2 to the replay counter, and proceeds to the next step.

> Note: The value 2^{48} was selected by the following reasoning. The proof of OCB mode security indicates the insecurity of the construction increases as $O(s^2/2^{128})$, where *s* is the total number of blocks protected. If *A* is the probability that an adversary can break the underlying block cipher AES, then the choice of $s = 2^{48}$ bounds chances of breaking AES-OCB mode to no more than approximately $A + (2^{48})^2/2^{128} = A + 1/2^{32}$; that is, using a single key in OCB mode over 2^{48} blocks does not greatly increase the adversary's chances over breaking a single block encrypted under AES. On the other hand, the replay counter is transmitted with the encrypted data, and it is necessary to minimize the number of bits transmitted through the wireless medium; further, it is desirable to use an odd number of bytes for the sequence number, so the existing WEP KeyID byte could be maintained to simplify hardware implementations. This limited the choices to 24-bits, 28-bits, 40-bits, 56-bits, *etc*. 24-bits is too small, but security decays too much with 56-bits. While 40-bits can be selected, it requires the counter to be interspersed in the replay sequence number field as the KeyID bits are in fixed bit positions 30 and 31. However, if we expand from 24-bits to 28-bits, it allows us to maintain a 32-bit replay sequence number field and enough blocks to be processes with a reasonable lifespan for the key .

8.3.3.4.3 Encoding the transmit Replay-Counter

The WRAP privacy algorithm Replay Counter is a four-octet field. It is used to convey the MSDU sequence number to the peer. The Replay Counter is utilized to construct the nonce and to detect replayed MSDUs.

The replay counter computed in 8.3.3.4.2 is encoded into the *Replay-Counter* field. This is accomplished by first encoding the number as a 28-bit big-Endian integer *BEI*. Next the three most significant bytes of *BEI* are encoded into the first three octets of the *Replay-Counter* field. Following these three octets the remaining 4-bits is concatenated with the 2 KeyID bits. Symbolically,

$BEI \leftarrow$ Big-Endian($replay\ counter \cdot 16$)
Partition *BEI* into a sequence of 4 octets: $BEI = BEI_1 \parallel BEI_2 \parallel BEI_3 \parallel BEI_4$, where
$B\ EI_4 = BEI_{bit25} \parallel BEI_{bit26} \parallel BEI_{bit27} \parallel BEI_{bit28} \parallel 0^4 KeyID \leftarrow 0^{68} \parallel keyid_{bit1} \parallel keyid_{bit2}$
$Replay\text{-}Counter \leftarrow BEI_1 \parallel BEI_2 \parallel BEI_3 \parallel KeyID$

This format matches the WEP IV field, with the exception of the use of the first nibble in the KeyID octet.

8.3.3.4.4 Construct the OCB nonce

This algorithm works for both transmit and receive. OCB mode requires a unique nonce be used for each message it encrypts for its security guarantees to be valid. Using the just-created *Replay-Counter* from clause 8.3.3.4.3, the implementation shall construct the OCB nonce as the concatenation of (a) the sequence number encoded as a big-Endian value, i.e., with its most significant bit first and least significant bit last, (b) its QoS traffic class, (c) the MSDU source MAC address, and (d) the MSDU destination MAC address:

$$nonce \leftarrow Replay\ Counter \parallel QoS\text{-}Traffic\text{-}Class \parallel Source\text{-}MAC\text{-}Address \parallel Destination\text{-}MAC\text{-}Address$$

If QoS traffic classes are not in use, the *QoS-Traffic-Class* value shall be 0^4, *i.e.*, 4 bits of zero. The *Source-MAC-Address*, *Destination-MAC-Address*, and *QoS-Traffic-Class* shall be encoded in the nonce in the same octet order as in their MSDU encoding. This nonce construction guarantees nonce unicity of these values. Notice *Source-MAC-Address* may differ from the IEEE 802.11 transmit address. Similarly, the *Destination-MAC-Address* may differ from the IEEE 802.11 receiver address.

> Note. It is feasible for an IEEE 802.11 implementation to construct a duplicate nonce by using the wrong station's MAC address as the source or destination MAC address, but such a construction is non-conformant. This can be a security problem for broadcast/multicast. If a deployment experiences a rash of duplicate nonces for broadcast multicast, it may indicates either a non-conformant implementation, a "traitor" within the BSS—*i.e.*, a party intentionally misbehaving—or a compromise of the BSS broadcast/multicast key.

8.3.3.4.5 Protect the transmit MSDU

The implementation shall use the WRAP temporal key T_K constructed in 8.6 and the *nonce* constructed in 8.3.3.4.4 to OCB encrypt the plaintext MSDU data. This results in two outputs:

a) An *OCB-ciphertext* string. This string contains the same number of octets as the MSDU plaintext data; and

b) A 64-bit *OCB-tag*.

Symbolically,

$$\textit{OCB-ciphertext} \,\|\, \textit{OCB-tag} \leftarrow \text{OCB-Encrypt}(T_K, \textit{nonce}, \textit{MSDU-data})$$

where OCB-Encrypt(A, B, C) denotes encrypting its third parameter C under key A using nonce B.

8.3.3.4.6 Construct the MSDU transmit payload

Finally, all the elements are assembled in the final MSDU payload. The WRAP privacy-protected MSDU payload consists of the concatenation of the *Relay-Counter* field (8.3.3.4.3), the OCB-ciphertext, and the OCB-tag (8.3.3.4.5):

$$\textit{MSDU-Data} \leftarrow \textit{Replay-Counter} \,\|\, \textit{OCB-ciphertext} \,\|\, \textit{OCB-tag}.$$

8.3.3.4.7 Receive context selection

The recipient shall select the appropriate context for the received MSDU based on the Transmit and Receive MAC addresses and the KeyID bits. If the Receive address is broadcast/multicast, then the selected context becomes the broadcast context. If not, the receiver verifies there is a unicast context for the frame. If the selected context is for the WRAP privacy algorithm, then the receiver continues with the AES-based privacy decapsulation algorithm.

If the WRAP privacy algorithm is utilized by an association, the receiver must treat all MSDUs as protected. Without this provision, an attacker can forge a valid message by simply sending a clear text message. Hence all implementations must maintain some state indicating whether WRAP privacy protection should be applied to received MSDUs, whether or not the WEP bit from the MAC header is asserted, and whether or not the *KeyID* bits are actually zero.

8.3.3.4.8 Receive sanity checks

If an applicable AES context is present, the receiver shall discard the received MSDU if it does not consist of at least 15 octets and increment the context's *dot11WrapFormatErrors* counter. This includes 3 octets of LLC header, and 12 octets of AES-based protocol overhead octets.

A second check is the total number of blocks. The implementation computes the total number of blocks protected in the MSDU. This is defined as

$$m = \lceil (\#\text{ MSDU data octets} - 12)/\text{AES-Block-Size} \rceil,$$

where $\lceil a \rceil$ means to round a up to the nearest integer, and AES-Block-Size = 16.. The 12 is removed to account for the MSDU *Replay Counter* field and the *OCB-tag* field.

If adding the number of blocks m will cause the value of *dot11WrapRecvdBlocks* from the context selected in 8.3.1.3.4.3 to wrap—i.e., if $m + \textit{dot11WrapRecvdBlocks} > 2^{48}$—then the cryptographic protection afforded by the key are considered exhausted, and it is a protocol error to use the key any further. The receiver shall discard the MSDU and increment the context's *dot11WrapSpentKeyErrors* counter.

8.3.3.4.9 Decrypting the MSDU data

Use the nonce constructed in 8.3.3.4.4 and the AES key from the context selected in 8.3.3.4.7 to OCB decrypt the received MSDU. By definition, this consists of

$$\textit{data-to-decrypt} \leftarrow \textit{MSDU-ciphertext} \parallel \textit{OCB-tag}.$$

The OCB decryption algorithm will result in two one of outputs:

a) A verification of the tag, and the decrypted plaintext;

b) Failure, because the decryption algorithm detected a change in the underlying data.

If the OCB decryption reports failure, the receiver shall increment the context's *802dot11AesDecryptErrors* counter, and discard the MSDU.

8.3.3.4.10 Unicast replay verification

If the received MSDU was unicast, the receiver also determines whether it is fresh or represents a replay. The receiver shall skip this step for broadcast/multicast MSDUs.

The MSDU sequence number is needed to provide replay protection. The little-Endian encoding of the MSDU sequence number can be extracted from the *Replay-Counter* field by dropping the last four bits of the *Key-ID* octet:

if $\textit{Replay-Counter} = RC_1 \parallel RC_2 \parallel RC_3 \parallel RC_4$ then
$\quad\text{Big-Endian}(\textit{SeqNum}) \leftarrow RC_1 \parallel RC_2 \parallel RC_3 \parallel (RC_4 \wedge 1^4 0^4)$

where "$\wedge$" denotes bit-wise AND. To determine whether a unicast represents a replay, the receiver shall test whether the MSDU replay counter *SeqNum* extracted from the MSDU *Replay Counter* field is a ***fresh*** value. It is fresh if the pair <*QoS-Service-Class*, *SeqNum*> has never been received in a valid MSDU for the context's key, and is declared a replay otherwise. If the MSDU's sequence number is a replay, the receiver shall discard the MSDU, increments the *dot11WrapReplays* counter, and halts the decapsulation. Note that the AES transmit encapsulation implies that MSDUs sent from the STA to the AP always use even values for the sequence number, and MSDUs sent from the AP to the STA always use odd values for the sequence number. Hence, the sequence number checking at an AP shall verify that the constructed *SeqNum* value is even, and at the STA that the constructed *SeqNum* value is odd; the implementation shall increment the *dot11WrapReplays* counter and halt the decapsulation of this check fails. The IEEE 802.11 implementation may use any suitable technique to guarantee that the pair <*QoS-Traffic-Class*, *SeqNum*> is fresh—e.g., it might maintain a sliding replay window, or it can maintain a list of all MSDU sequence numbers correctly received, *etc*.

8.3.3.4.11 Completing reception

If the MSDU has not been discarded due to the processing described above, then the receiver must update the *802dot11RecvdBlocks* counter by adding to it the value *b* computed in 8.3.3.4.2, to indicate the number of blocks decapsulated, and the decapsulation completed successfully.

8.3.4 The Counter-Mode/CBC-MAC protocol (CCMP)

A protocol based on the ***Advanced Encryption Standard*** (AES) and ***Counter-Mode/CBC-MAC*** (CCM) mode has been adopted. This protocol is called the *Counter-Mode/CBC-MAC Protocol* (*CCMP*), and this clause defines it. Implementation of this protocol is mandatory for RSN compliance.

8.3.4.1 CCMP overview

The CCMP protocol is based on AES using the CCM mode of operation. The CCM mode combines ***Counter*** (CTR) mode privacy and ***Cipher Block Chaining Message Authentication Code*** (CBC-MAC) authentication. These modes have been used and studied for a long time, have well-understood cryptographic properties, and no known patent encumbrances. They provide good security and performance in both hardware or software.

CCM uses the same temporal key for both CTR mode and the CBC-MAC. Using a key for more than one function usually introduces a weakness. Jakob Jonsson has proved that this cannot occur in this particular case, as the construction of different IVs for CTR-mode and CBC-MAC eliminates the problems usually associated with this. Indeed, all the encryption IVs are different, and they are different from the authentication initial block. If the block cipher behaves like a random permutation, then the outputs are independent of each other, up to the insignificant limitation that they are all different. The only places where the inputs to the block cipher can overlap is an overlap between an intermediate value in the CBC-MAC and one of the other encryptions. As all the intermediate values of the CBC-MAC computation are essentially random (because the block cipher behaves like a random permutation) the probability of such a collision is very small. Even if there is a collision, these values only affect MIC, which is encrypted so that an attacker cannot deduce any information, or detect any collision.

CCM assumes a fresh temporal key for every session. Reuse of a temporal key and packet number voids all security guarantees.

Annex F provides test vectors for CCM mode.

8.3.4.1.1 CCMP encapsulation

Figure 21 depicts the CCMP encapsulation process. CCMP encapsulates a plaintext MPDU using the following steps:

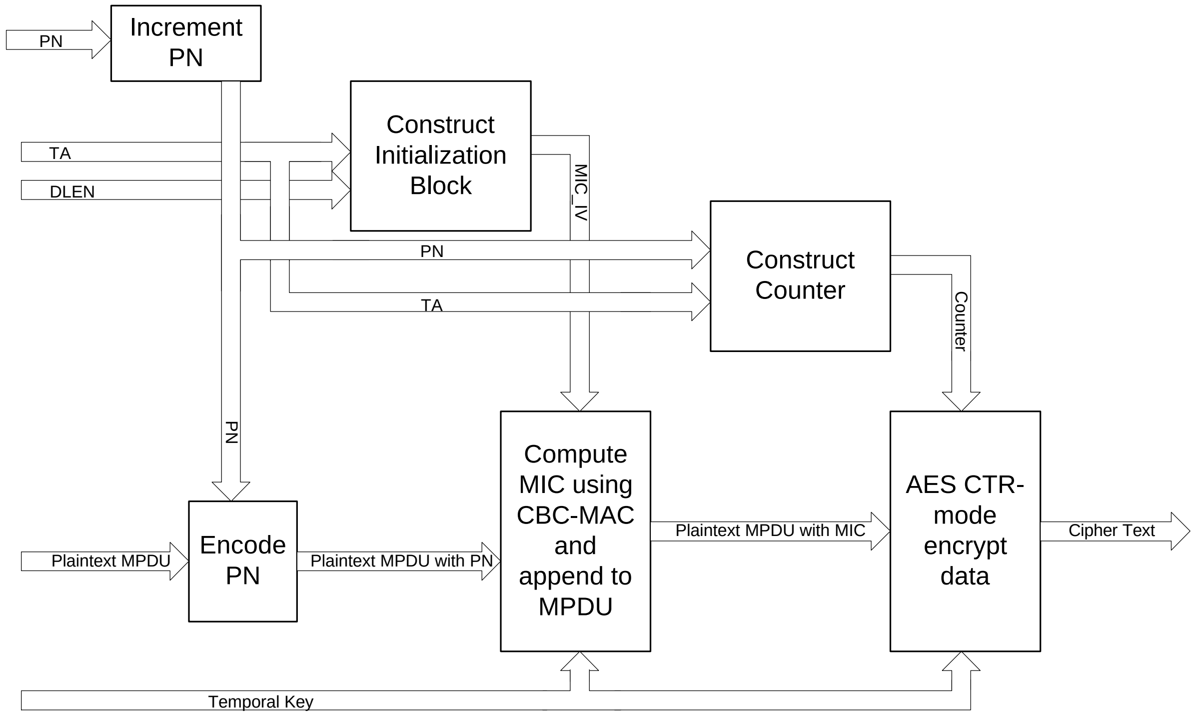


Figure 21—CCMP encapsulation block diagram

1. It first increments the Packet Number (PN), to obtain a fresh PN for each MPDU.
2. It encodes the fresh PN into the MPDU.
3. It constructs the CCM initial block from the PN, the MPDU TA, and from the MPDU data length (Dlen).
4. With the initial block constructed, it MICs the MPDU using AES with CBC-MAC.
5. It constructs the CCM CTR-mode counter from the PN and the MPDU TA.
6. Finally, it encrypts the MPDU data and MIC using AES in CTR-mode.

8.3.4.1.2 CCMP decapsulation

Figure 22 depicts the CCMP decapsulation process. CCMP decapsulates a plaintext MPDU using the following steps:

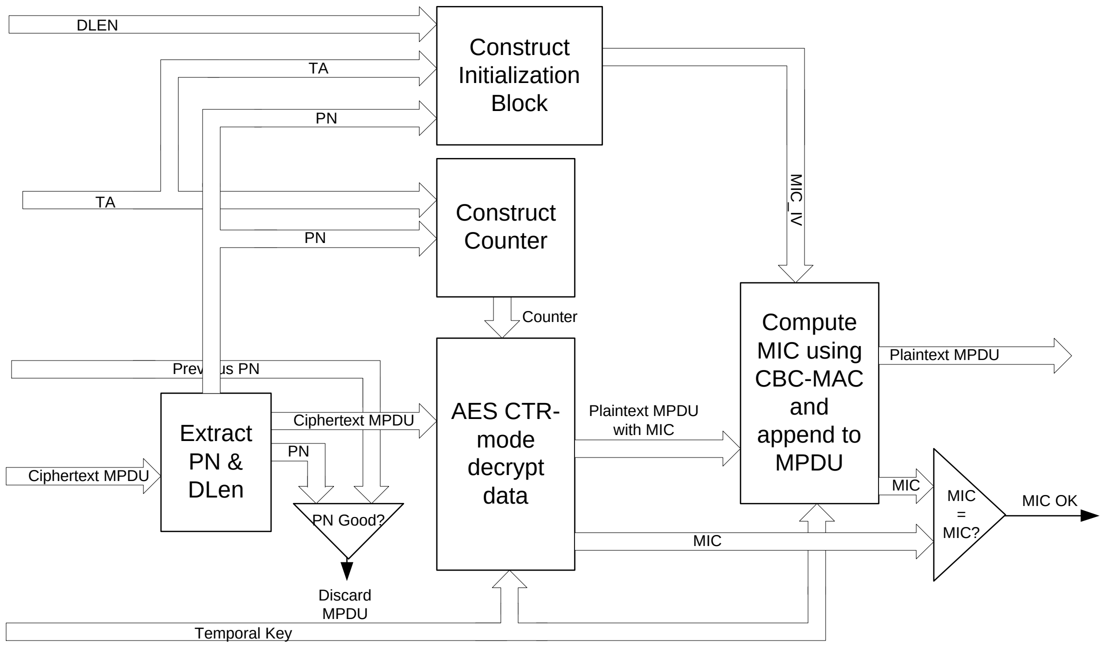


Figure 22—CCMP decapsulation block diagram

1. It first decodes the PN and Dlen.

    Informative Note: The PN can be removed from the MPDU at this or any other step.

    Informative Note: Dlen must be at least eight (16) octets, to account for the MIC and the encoded PN.

2. It applies replay filtering. If the PN indicates out-of-sequence arrival, the MPDU is discarded as a replay.

3. The CCM CTR-mode counter is constructed from the TA and PN.

4. The counter and Temporal key are used to CTR-mode decrypt the MPDU data. Note this operation is the same as CTR-mode encryption.

5. It constructs the initial block used to form the CCM CBC-MAC IV from the PN, TA, and Dlen.

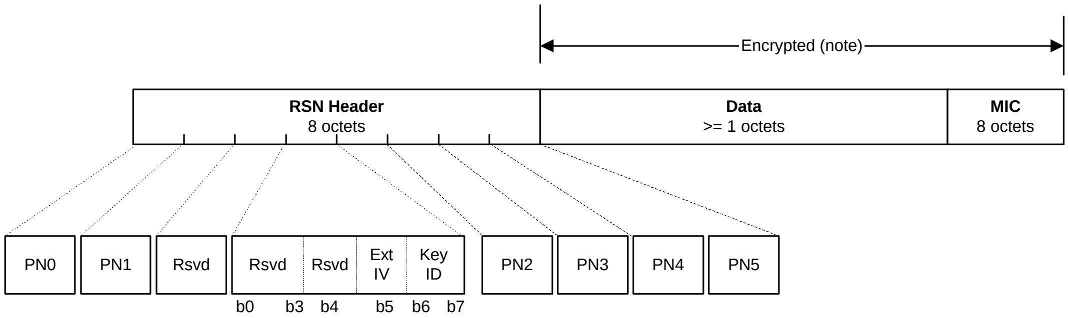
    Informative Note: Dlen must be decremented by sixteen (16) octets, as the MIC and the encoded PN are not considered part of the plaintext data being protected.

6. It uses the initial block and temporal key to re-compute a MIC′ of the decrypted MPDU, using AES with CBC-MAC.

7. It finally compares the MIC′ it computed with the received MIC. If the two do not match, the MPDU is discarded as a forgery.

8.3.4.2 CCMP MPDU format

Figure 23 depicts the MPDU when using CCMP.



Note: The encipherment process has expanded the original MPDU size by 16 octets, 4 for the PN0-1 / Key ID field, 4 for the PN2-5 field and 8 for the Message Integrity Code (MIC).

Figure 23—Expanded CCMP MPDU

The IV/KeyID and Extended IV fields together are called the ***encoded PN***. This is a slight abuse of language, since the encoding includes the Key Id as well as the PN.

The CCMP formats are invisible to entities outside the IEEE 802.11 MAC data path.

Bit 5 of the KeyID octet signals an Extended Packet number field of 6 octets. For standard length Packet number/ IV fields this bit shall be set to zero (0), for extended packet number field the bit shall be set to one. The Extended IV bit (bit 5) is always set for CCMP.

The reserved bits shall be set to zero (0).

8.3.4.3 CCMP state

CCMP privacy uses a MIB array called the *dot11CcmpKeyMappings*. This supports zero, one, or two entries for each MAC address pair with which the STA maintains secure associations. The size of the *dot11CcmpKeyMappings* array is implementation-specific. A global MIB variable *dot11CcmpKeyMappingLength* indicates the number of entries in the array.

Each entry of the *dot11CcmpKeyMappings* groups together the following state:

1. A *dot11CcmpReceiveAddress* and a *dot11CcmpTransmitAddress*, indicating that this entry applies to all MPDUs being sent between this pair of addresses.

2. A *dot11CcmpKeyID*, indicating the KeyID into which this entry maps.

3. A 128-bit key called the *dot11CcmpTemporalKey*, referred to informally as the temporal key. This is the TK1 subfield portion of the Pairwise Transient Key as defined in 8.5.1.2, or the TK1 subfield of the Group Transient Key as defined in 8.5.1.3. This key is often called the temporal key.

4. A set of 48-bit counters called the *dot11CcmpTrafficClassNPacketNumber*, for constructing the next initial block. *N* ranges from 0 to 15, with one traffic class defined for each QoS service class. When QoS is not used, only *dot11CcmpTrafficClass0PacketNumber* is used.

5. A set of 48-bit replay windows called the *dott11CcmpTrafficClassNReplayWindow*, for detecting replays. *N* ranges from 0 to15. When QoS is not used, only *dot11CcmpTrafficClasse0ReplayWindow* is used.

6. A boolean flag called *dot11CcmpEnableTransmit*, to indicate when the temporal key can be used for transmitting MPDUs.

7. A boolean flag called *dot11CcmpEnableReceive*, to indicate when the temporal key can be used for receiving MPDUs.

8. A 32-bit counter *dot11CcmpFormatErrors*, to indicate the number of MPDUs received with an invalid format, *initializ*ed to zero.

9. A 32-bit counter *dot11CcmpReplays*, to indicate the number of received unicast MPDUs discarded by the replay mechanism, *initializ*ed to zero.

10. A 32-bit counter *dot11CcmpDecryptErrors*, to indicate the number of received MPDUs discarded by the CCMP decryption mechanism, *initializ*ed to zero.

11. A 48-bit counter *dot11CcmpRecvdMPDU*, to track the total number of protected MPDUs received.

Informative Note 1: A broadcast/multicast entry does not utilize the replay window. This is because it is impossible to detect broadcast/multicast replays using symmetric key techniques. In particular, any party holding the broadcast/multicast key can masquerade as any other member of the group, so can intrude on another's sequence space without detection.

Informative Note 2: As an optimization, implementations may compute and maintain the AES-CCM key schedule rather than maintain the temporal key.

8.3.4.4 CCMP procedures

8.3.4.4.1 Increment the PN

This procedure increments the Packet Number (PN) by 1:

$$PN \leftarrow PN + 1$$

such that the resulting PN < 2^{48}.

Informative Note: When the PN space is exhausted, the choices available to an implementation are to replace the temporal key with a new one, to end communications, or to send further traffic unprotected. Reuse of any PN value compromises already sent traffic. The PN is large enough, however, that PN space exhaustion should not be an issue.

8.3.4.4.2 CCM initial block construction

Informative Note: CCM is a big-Endian algorithm. This section therefore explicitly represents data structures as big-Endian quantities instead of the conventions of 7.1.1.

The CCM initial block shall have the format

Bit Number within field	B7 B0	B103												B0	B7 B0	
Octet Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Contents	Flag	Nonce	Dlen

Figure 24—CCM Initial Block Format

Here

- The Flags field occupies bits 127-120 of the CCM Initial Block. Flags is a bit field assuming the value 0x59 (hex). The bits shall be interpreted as follows:
    - 7: reserved: value = 0
    - 6: Include header: value = 1, meaning yes
    - 3-5: MIC size: value = 3, meaning use an 8-octet MIC
    - 0-2: Dlen size: value = 1, meaning use a 2-octet Dlen

bit:	B7	B6	B5 B3	B2 B0
	0	1	0 1 1	0 0 1

Figure 25—CCM Initial Block: Flag Field

- The Nonce field occupies bits 119-16 of the CCM Initial Block. The Nonce has an internal structure QoS-TC || A2 || PN, where
    - QoS-TC occupies bits 103-96 of the Nonce (bits 119-112 of the Initial Block). This field is reserved for the QoS traffic class and shall be set to the fixed value 0 (0x00 hex).
    - MPDU address A2 occupies bits 95-48 of the Nonce (bits 111-64 of the Initial Block). This shall be encoded with the octets ordered with A2 octet 0 at octet index 2 and A2 octet 5 at octet index 7.
    - PN occupies bits 47-0 of the Nonce (bits 63-16 of the Initial Block). This field shall encode the MPDU sequence number associated with the temporal key. The octets of PN shall be ordered such that PN0 is at octet index 13 and PN5 is at octet index 8.

Octet Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Content	0x00	A2						PN					

Figure 26—CCM Initial Block: Nonce Field

- The Dlen field occupies bits 15-0 of the CCM Initial Block. Dlen represents the length of the plaintext MPDU length in octets. This shall be encoded using the reverse bit ordering from the usual conventions from 7.1.1, with the Dlen msb first and the Dlen lsb last.

  Informative Note: Dlen is the length of the data proper, and does not include the length of the MIC, nor of the encoded PN.

Informative Note: The initial block construction was chosen to permit the same temporal key to be used for both encryption and the MICing operation, and to protect traffic in both directions over an IEEE 802.11 link.

8.3.4.4.3 CCMP MIC computation

CCMP uses AES in the CBC-MAC mode to compute a MIC for the MPDU.

The input to this algorithm is

1. The plaintext MPDU.
2. The Initial Block for this MPDU, as constructed in 8.3.4.4.2.
3. The temporal key. 8.6.5 defines this key for pairwise communication, and 8.6.6 defines this key for group communications.

The output of the algorithm is a MIC value. This can be appended to the MPDU on transmit, and compared with a received MIC at the receiver.

The algorithm first encrypts the Initial Block to produce the CBC mode IV. Next it computes the CBC-MAC over the IEEE 802.11 header length (Hlen), selected parts of the IEEE 802.11 MPDU header, and the plaintext MDPU data.

The algorithm represents the header length Hlen as a big-Endian (i.e., msb first) unsigned integer value. The algorithms decrements the genuine Hlen by 2 (length of the omitted duration field) prior to encoding.

When the number of octets in the Hlen together with the parts of the IEEE 802.11 header protected is not a multiple of the AES block size, the header data shall be zero padded to a multiple of the AES block size (16 octets). This padding is used only by the algorithm, and is not included in the transmitted MPDU.

Informative Example. When A3 is not present, for instance, the total data being protected by the MIC is 18 octets. In this case 14 zero octets are appended to the header data for the MIC computation and then discarded.

The portions of the header included in the computation include

- FC – MPDU Frame Control field, with Retry bit masked to zero.
- A1 – MDPU Address 1.
- A2 – MPDU Address 2..
- A3 – MPDU Address 3, if present.
- A4 – MPDU Address, if present.
- SC – MPDU Sequence Control.
- QC – The Quality of Service Control, if present.

Informative Note: The algorithm skips the header Duration field, because it value is mutable, i.e., it can change due to normal IEEE 802.11 operation. Similarly, the computation masks the FC Retry bit to zero, as the value of this bit is mutable.

Informative Note: In spite of being mutable, the MIC computation includes the Sequence Control field. This is CCMP's means of defending against fragmentation attacks. Fragmentation attacks against the protocol are always possible, given that CCMP protects MPDUs instead of MSDUs.

When the MPDU plaintext data is not a multiple of the AES block size, zero padding shall be added to extend the plaintext data length to be the first multiple of the AES block size larger than the real length. This padding is present only for the computation, and shall not be part of the transmitted data.

Informative Examples. If the plaintext data field consists of 96 octets, no padding is require as 96 = 6·16. If the plaintext data field length consists of 100 octets, then 12 octets of zero padding are appended to the plaintext data for the MIC computation and then discarded once it completes.

The CBC-MAC computation produces in a 128-bit tag value. CCMP truncates the tag to its most significant 64 bits (bits 127-64) to form the MIC. Figure 27 depicts the entire process for an example MPDU with an arbitrarily chosen payload length of 58 octets.

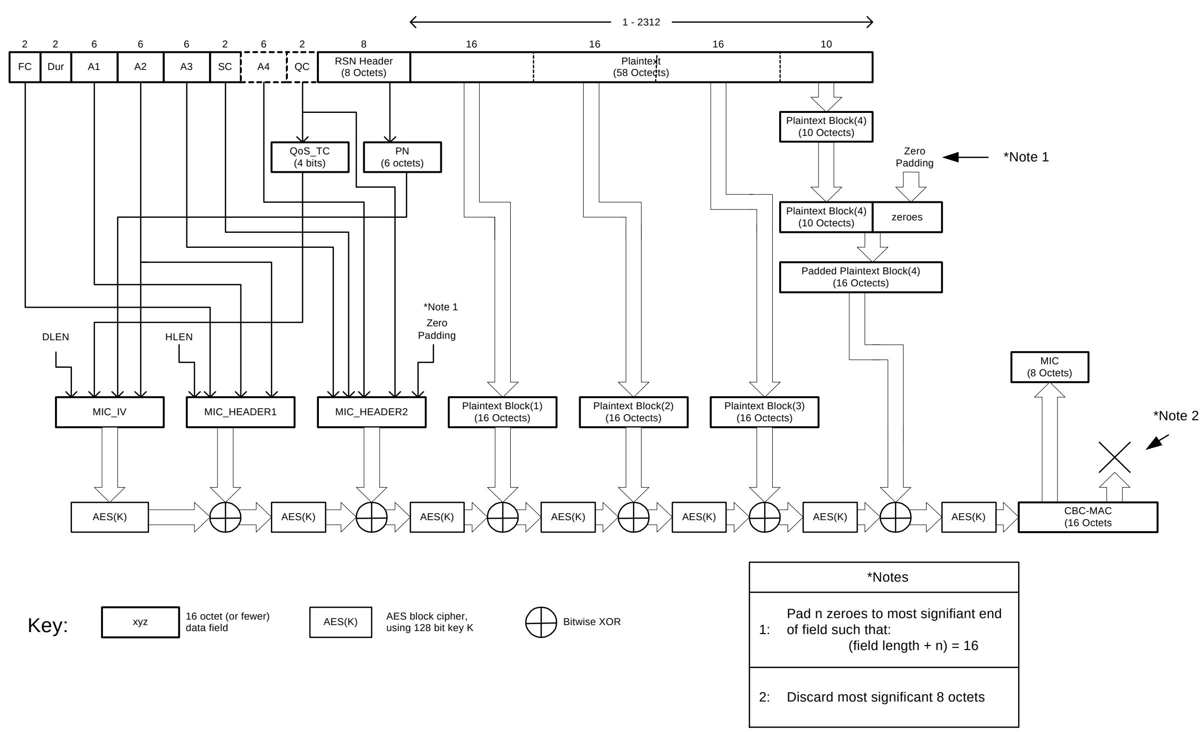


Figure 27—CCMP MIC computation block diagram

The construction of the MIC_IV referred to in Figure 27 is summarized below in Figure 28.

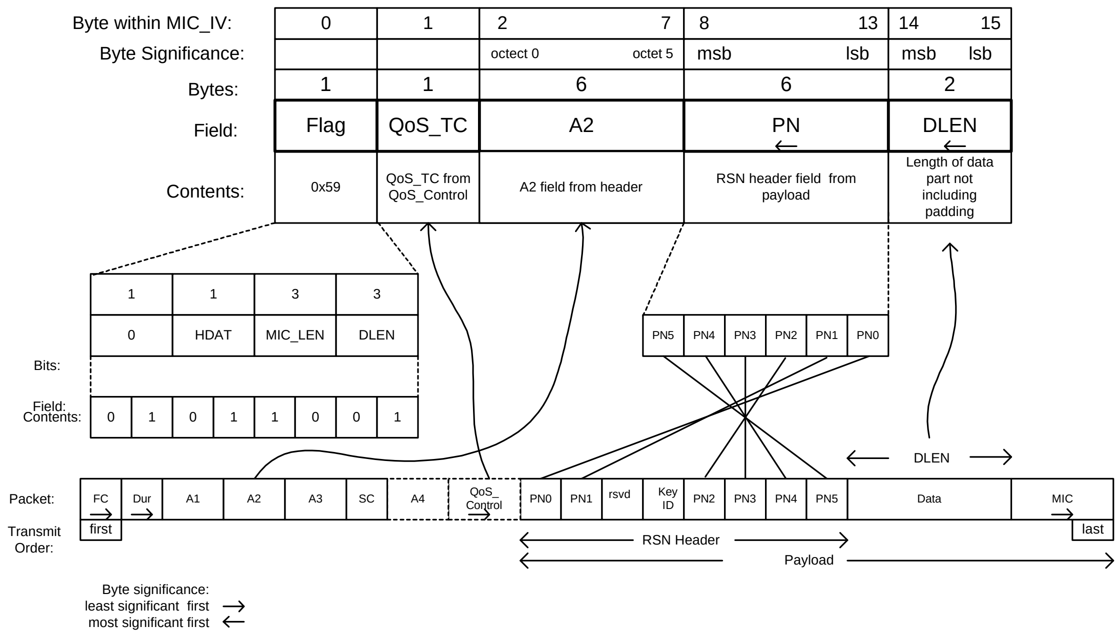


Figure 28—MIC IV Construction

The second block to be included in the MIC computation is formed from fields within the MPDU header. The construction of this block is summarized in Figure 29.

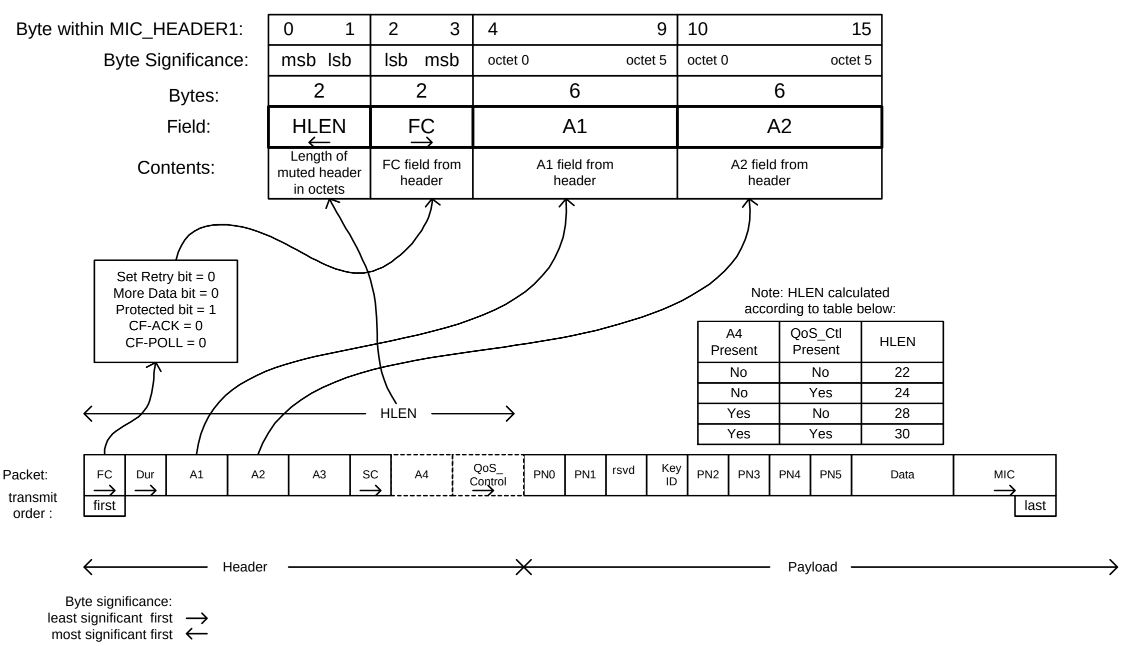


Figure 29—MIC HEADER1 Construction

The third block to be included in the MIC computation is formed from fields of the header, if necessary with padding to bring the block size to 16 octets. The construction of the third MIC generation header block is summarized in Figure 30.

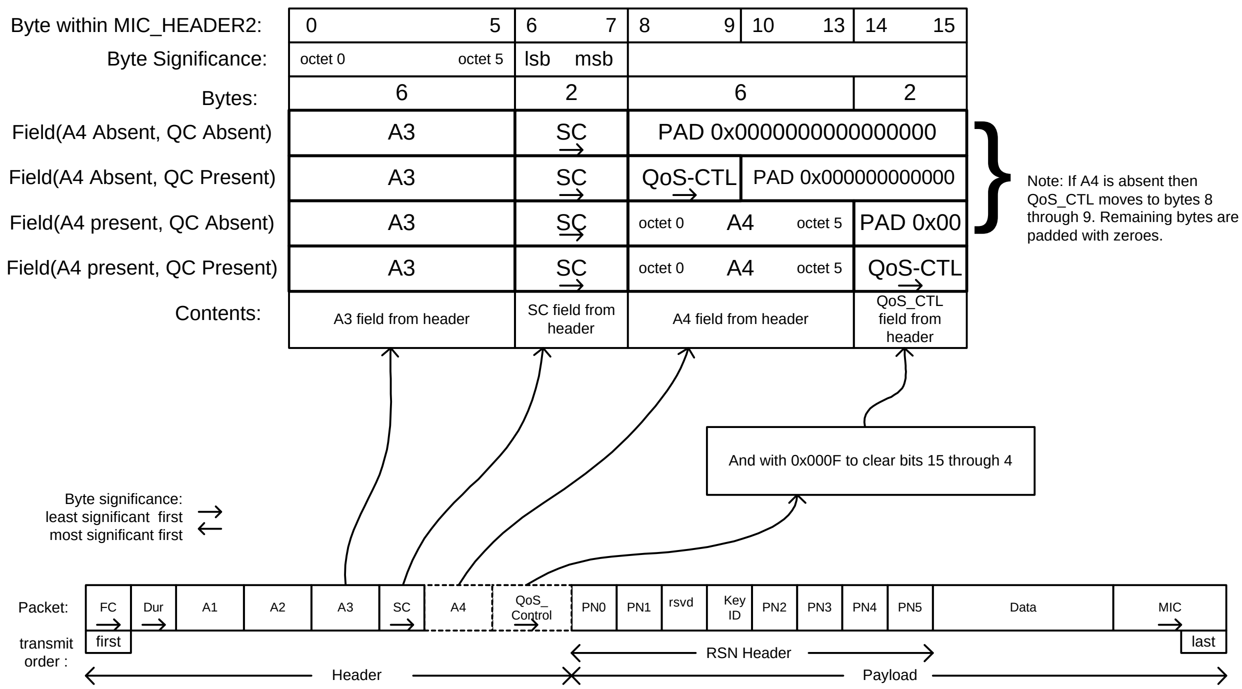


Figure 30—MIC HEADER2 Construction

8.3.4.4.4 CCMP MIC verification

On transmit the MIC computed in 8.3.4.4.3 is appended to the plaintext MPDU data. Thus, the MIC becomes the final 64-bits of plaintext MPDU data. Note that CCMP appends the MIC to the plaintext data prior to encryption.

To verify the MIC, after decrypting the data, the receiver computes the MIC using the procedure in 8.3.4.4.3 and bit-wise compares the result against the last 64-bits of plaintext MPDU data. If any bits of the computed MIC differ from those received, it discards the MPDU as a forgery. If all of the bits are identical, then the receiver interprets the MPDU as genuine, and strips the 64-bit MIC from the end of the MPDU.

8.3.4.4.5 CCM CTR-mode counter construction

Informative Note: CCM is a big-Endian algorithm. This section therefore explicitly represents data structures as big-Endian quantities instead of the conventions of 7.1.1.

The CCM CTR-mode counter shall have the format

Octet Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Content:	Flags	Nonce													Ctr	

Figure 31—CCM Counter Format

Here

- The Flags field occupies bits 127-120 of the counter. Flags represents a bit field assuming the value 0x01 (hex). The bits shall be interpreted as follows:
    - o 7: reserved: value = 0
    - o 6: Include header: value = 0, meaning no
    - o 3-5: MIC size: value = 0, meaning none
    - o 0-2: counter size: value = 1, meaning use a 2-octet C field

bit:	B7	B6	B5 B3	B2 B0
	0	0	0 0 0	0 0 1

Figure 32—CCM Counter: Flags Field

- The CCM Counter Nonce format is identical to that for the CCM Initial Block, defined in 8.3.4.4.2.
- Ctr represents the lower 16-bits of the CTR-mode counter, and takes the value of 0x0000.

  Informative Note. The counter format permits CTR-mode to be used with MPDU plaintext payloads of up to $(2^{16} - 1)\cdot 16 + 8 = 1048568$ octets in length.

Informative Note: The counter construction was chosen to permit the same temporal key to be used for both encryption and the MICing operation, and to protect traffic in both directions over an IEEE 802.11 link.

The detailed construction of the CCM counter is described below in Figure 33.

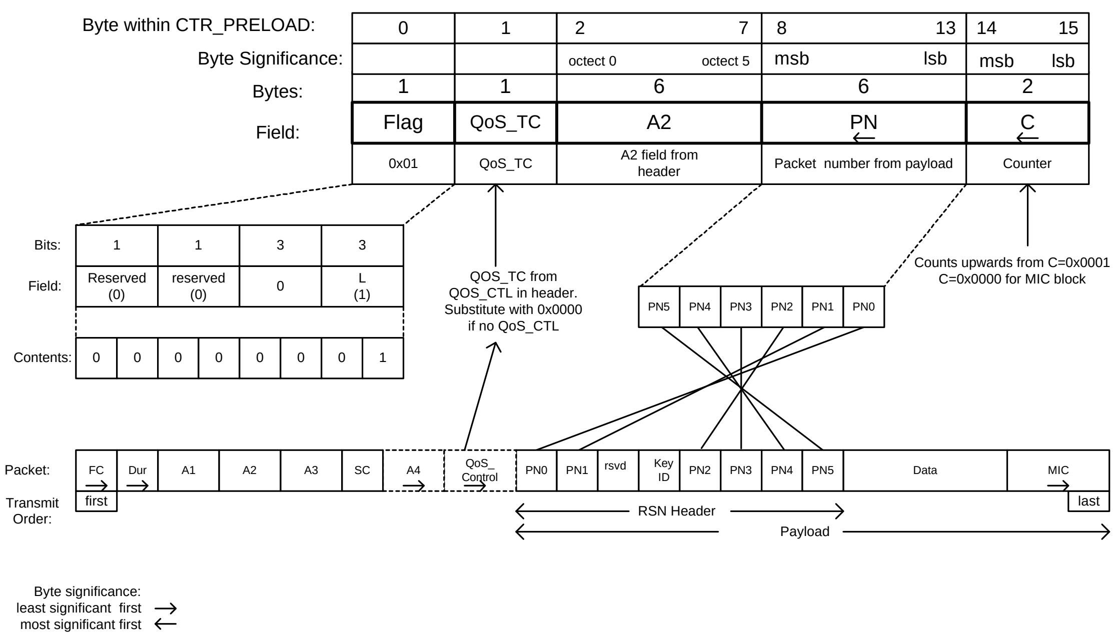


Figure 33—CCM Counter Nonce Construction

8.3.4.4.6 CCM CTR-mode encryption

CCMP uses AES in Counter mode to encrypt and decrypt the MPDU data and MIC.

The input to this algorithm is

1. The MPDU data field, with MIC appended. On transmission, the data field with MIC is plaintext, while on reception bother are ciphertext.
2. The Counter for this MPDU, as constructed in 8.3.4.4.5.
3. The temporal key. 8.6.5 defines this key for pairwise communication, and 8.6.6 defines this key for group communications.

The output of the algorithm is an encrypted MPDU data field on transmit and a decrypted MPDU data field with MIC on reception.

Figure 27 depicts the encryption process for an example MPDU with an arbitrarily chosen payload length of 58 octets. Figure 35 depicts the decryption process for the same MPDU.

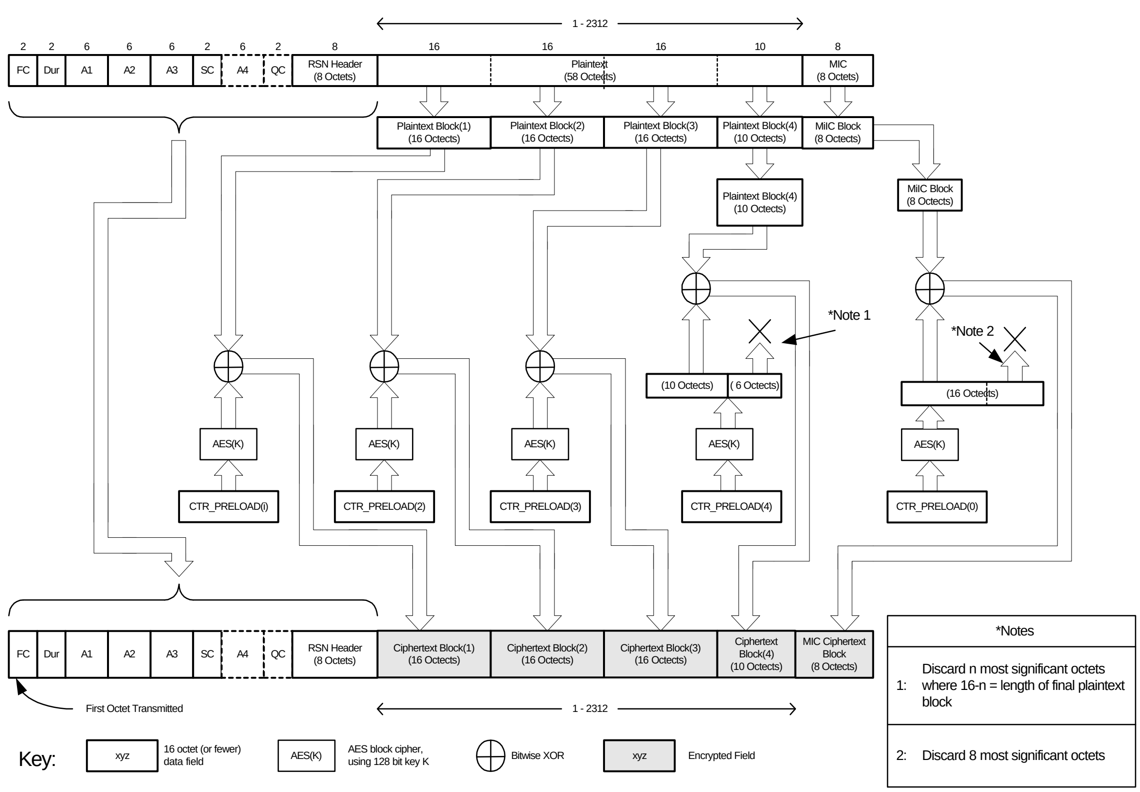


Figure 34—CCMP CTR-mode encryption block diagram

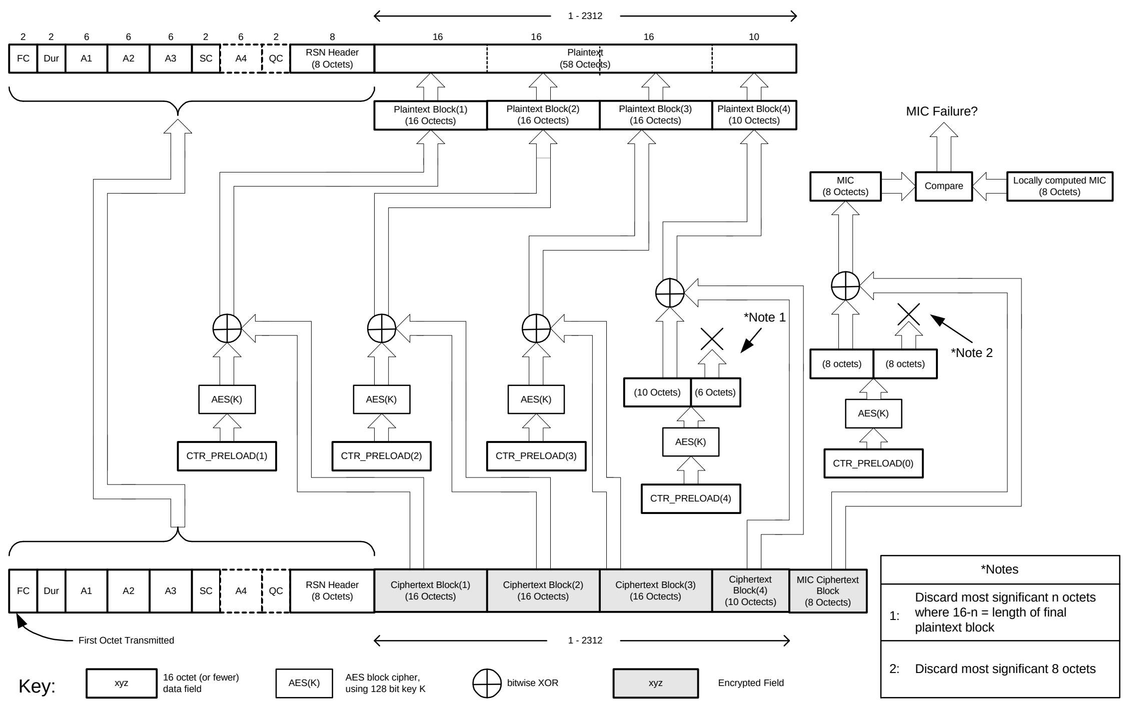


Figure 35—CCMP CTR-mode decryption block diagram

Counter mode operates by encrypting a counter value. The data field is partitioned into contiguous blocks $D_1 \ldots D_n$ each of whose length equals the AES block size (16 octets); the final block D_n may be shorter if the entire data field is not a multiple of the block size. Each Di is then encrypted (decrypted) as

$$\text{Counter.Ctr} \leftarrow \text{BigEndian}(i);\ \text{AES\_Encrypt}_K(\text{Counter}) \oplus D_i$$

where "$\oplus$" denotes the exclusive OR operation, $\text{AES\_Encrypt}_K(\cdot)$ denotes AES encryptions of its argument under the key K, and BigEndian(i) denotes the big-Endian (msb first) encoding of its argument as a unsigned integer. The key K denotes the temporal key associated with the current security association.

Informative Note: If the final block to be encrypted (decrypted) is not a multiple of the AES block size (16 octets), then the final encrypted counter value is truncated to match the length of the final block; since the Counter is Big-Endian, the least significant bytes are dropped. In particular, counter mode requires no padding.

Informative Note: Because an IEEE 802.11 MPDU can convey 0-2304 octets of data, this implies that a CCMP protected MPDU will convey between 1 and 145 blocks of encrypted data. Thus the value of i above ranges from 1 to n, where $n \leq 145$.

Informative Note: Encrypting the MIC avoids collision attacks on the CBC-MAC. If the block cipher behaves as a pseudo-random permutation then the key stream is indistinguishable from a random string. This implies that the attacker gets no information about the CBC-MAC results. The only known avenue of attack that is left is a differential-style attack, which has no significant chance of success if the block cipher is a pseudo-random permutation.

8.3.4.4.7 Encoding the PN

The PN is encoded in the IV/Key ID and Extended IV fields. When the PN is represented according to the conventions of 7.1.1, bits 0-7 of the PN are encoded as IV0, bits 8-15 as IV1, bits 16-23 as IV2, bits 24-31 as IV3, bits 32-39 as IV4, and bits 40-47 as IV5.

8.3.4.4.8 CCMP replay detection

1. CCM Packet Number (PN) values shall correspond to MPDUs.

2. The PN (48 bit counter) shall be selected from a single pool by each transmitter for each temporal key. Each transmitter has its own unique counter for each temporal key established.

3. The PN shall be implemented as a 48-bit monotonically incrementing counter, *initializ*ed to zero when the corresponding CCMP temporal key is *initializ*ed or refreshed.

4. The CCMP format carries the least significant 16 bits of the 48-bit PN. The remainder of the PN is carried in the Extended IV.

5. The recipient shall maintain a separate replay window for each IEEE 802.11 Traffic Class, and shall use the PN recovered from a received frame to detect replayed frames. A replayed frame occurs when the PN extracted from a received frame is repeated or not greater than the current Traffic Class replay window value for the frame's traffic class. The replay window accommodates frames that may be delayed due to traffic class priority values.

6. A receiver shall maintain a separate set of PN replay windows for each MAC address it receives CCMP traffic from. The receiver initializes the replay window whenever it resets the temporal key for a peer.

7. In order to accommodate burst ACK, the CCMP receiver shall check that the received PN (48 bit counter) is no smaller than 15 less than the greatest CCMP replay window value for the MPDU's temporal key. When combined with the prohibition on correctly decrypting more than one MPDU under a given <temporal key, PN> pair, this provides replay protection and accommodates frames that may be delayed due to message class priority values, with a window size of 16.

8.4 RSN security association management

8.4.1 Security association life cycle

IEEE 802.11 uses the notion of a security association to describe secure operation. Secure communications are possible only within the context of a security association, as this is the context providing the state—cryptographic keys, counters, sequence spaces, etc.—needed for correct operation of the IEEE 802.11 cipher suites.

The life cycle of a security association is naturally intertwined with the other IEEE 802.11 mechanisms. A STA can operate in either an ESS or in an IBSS, and a security association has a distinct life cycle for each.

In an ESS there are two cases: initial contact between the STA and the ESS, and roaming by the STA within the ESS. A STA and AP establish an initial security association via the following steps:

1. The STA selects an authorized ESS by selecting among APs that advertise an appropriate SSID.

   Informative note: Advertising the SSID plays a crucial security function. If the STA does not know the SSID of some AP, it either must decline communication, or it has to guess the ESS of the AP. When the AP is not authorized, then the STA might present all of its credentials in an effort to find some that allow it to authenticate. This can result in unintended identity disclosure of the STA to the unauthorized AP.

Advertising the SSID also provides an important performance optimization. Without advertisements, if the AP is indeed authorized, the STA on average must present half its credentials before locating the correct ones at initial contact.

2. The STA may then use IEEE 802.11 Open System Authentication followed by association to the chosen AP. Negotiation of security parameters takes place during association.

   Informative Note: An attack altering the security parameters will be detected by the key derivation procedure.

   Informative Note: IEEE 802.11 Open System Authentication provides no security, but is included to maintain backward compatibility of the state machine.

3. After the association completes, the STA and AP shall initiate filtering of non-IEEE 802.1X class 3 MPDUs, and the AP's Authenticator shall initiate IEEE 802.1X authentication. The authentication will be mutual, as the STA needs assurance that the AP belongs to the authorized network and is not a rogue.

   Informative Note: Any secure network cannot support promiscuous association as in unsecured operation of IEEE 802.11. A trust relationship must exist between the STA and the target SSID prior to association and secure operation, in order for the association to be trustworthy. The reason is that an attacker can deploy a rogue access point just as easily as a legitimate network provider, so some sort of prior enrollment procedure is necessary to establish credentials between the ESS and the STA.

4. The last step is key exchange. The authentication process creates cryptographic keys shared between the IEEE 802.1X AS and the STA. The AS distributes these keys to the AP, and the AP and STA use two key confirmation handshakes, called the 4-way handshake and group key handshake, to complete security association establishment. The key confirmation handshakes indicate when the link has been secured by the keys, so is safe to allow normal data traffic. If key handshakes complete successfully, STAs (including APs) shall terminate the filtering of class 3 MPDUs other than IEEE 802.1X, allowing normal data to flow.

   Informative note: The Supplicant of a STA should silently discard IEEE 802.1X messages not received from the AP.

A STA roaming within an ESS establishes a new security association by one of two schemes:

1. (Re-)Associating followed by IEEE 802.1X authentication. In this case the station repeats the same actions as for an initial contact association, but it also uses the MLME-DELETEKEYS.request to remove the cryptographic key from the IEEE 802.11 MAC when it roams from the old AP. The STA also deletes the cryptographic keys when it disassociates/deauthenticates from all BSSIDs in the ESS.

2. A STA already associated with the ESS can instead request its IEEE 802.1X Management Entity to authenticate with a new AP before associating to that new AP. In this case the Management Entity can request its IEEE 802.1X Supplicant to send an AuthenticationRequest to an AP with which it is not associated. The normal operation of the DSS via the old AP provides the communication between the STA and the new AP. The STA's IEEE 802.11 Management Entity delays Reassociation with the new AP until IEEE 802.1X authentication completes via the DSS. If IEEE 802.1X authentication completes, then cryptographic keys shared between the new AP and the STA will be installed, creating an environment where Reassociation without a subsequent IEEE 802.1X full authentication makes sense.

The MLME-DELETKEYS.request terminates a security association on the local STA. This primitive destroys the cryptographic keys established for the security association, so that they cannot be used to protect further IEEE 802.11 traffic. A STA's IEEE 802.11 Management Entity uses this primitive in one of two situations: when it disassociates or deauthenticates from an AP in an ESS, and when it associates to a new AP.

The life cycle of a security association is different in an IBSS. When explicit authentication is not used, a STA sets the AuthenticationRequest variable to request that its IEEE 802.1X implementation initiate the 4-way handshake of 8.5 with a Pre-Shared Key (PSK) with each IBSS peer STAs it encounters. A STA should use this primitive when it encounters another STA belonging to the IBSS with which it has no security association.

> Informative Note: A STA can receive IEEE 802.1X messages from a previously unknown MAC address. Membership in the IBSS is determined by the peer STA's ability to use the correct PSK.

> Informative Note: Any STA targeted from the IBSS may decline to form a security association with the joining STA. An attempt to form a security association may also fail because, e.g., the peer uses a different pre-shared key from that which the STA expects.

In an IBSS each STA defines its own group key to secure its broadcast/multicast transmissions. After establishing a security association, each STA shall use the Group Key Handshake to distribute its transmit Group Key to its new peer STA.

A security association terminates in an IBSS in the same way it does in an ESS, by the IEEE 802.11 Management Entity invoking the MLME-DELETEKEYS.request primitive.

> Informative Note: A STA should remove all association state and send a deauthenticate message if it receives an MLME-DELETEKEYS.request.

8.4.1.1 IEEE 802.11 ESS authentication and key management primer (Informative)

There are three authentication and key management architectures in IEEE 802.11, namely "Open System" and "Shared Key", which were defined for use in the context of WEP in IEEE 802.11-1999, and the newer IEEE 802.1X-based authentication mechanisms that are defined for use in the context of a Robust Security Network (RSN). In fact, the terms RSN and an IEEE 802.11 LAN using IEEE 802.1X and CCMP, WRAP, or TKIP are synonymous.

IEEE 802.1X "Port-Based Network Authentication" was originally designed for switched networks, in which eavesdropping is at least somewhat challenging. The original IEEE 802.1X standard assumes that tapping in to the communication link between a station and a switch is non-trivial, and relatively easy to detect. When the standard first appeared, networks were rapidly adopting switched topologies, abandoning shared hubs, so there was no strong demand for IEEE 802.1X to support shared-media LANs, although the standard does not prohibit operation over shared LAN topologies. As IEEE 802.11 LANs increased in popularity, the need for a proper authentication and key management presented itself, and it was natural to want to leverage mechanisms already been defined in another IEEE 802 standard.

IEEE 802.1X-2001 defines a framework based on the Extensible Authentication Protocol (EAP)[1] over LANs, also known as EAPoL. IEEE 802.1X extensions need to be defined to ensure that the network authentication services are secure in shared-medium networks such as those based on IEEE 802.11.

EAPoL is used to exchange EAP messages. EAP messages perform authentication between a STA and an EAP entity known as the Authentication Server (AS). A STA seeking to be authenticated uses EAPoL to communicate with a device that enforces the authentication, for example, an Ethernet switch or an IEEE 802.11 AP. The EAPoL exchange takes place between two entities, one associated with the station desiring to be authenticated, known as the "Supplicant," and the other associated with the device that enforces the access to the network, e.g., the switch or AP, known as the "Authenticator". Besides restricting network access only to authenticated stations, the Authenticator also acts as a mediator in the EAP conversation between the EAP Client and the AS.

[1] The EAP was originally designed to support authentication over the Point-to-Point Protocol (PPP), and is a product of the Internet Engineering Task Force (IETF).

EAP packets are encapsulated in EAPoL frames to enable them to cross the LAN medium. EAPoL also provides some control features—e.g., an EAPoL-Start message was defined to initiate authentication; similarly, an EAPoL-Logoff message was defined to terminate a connection. IEEE 802.1X-2001 also defined an optional capability to use the EAPoL-Key message to exchange encryption keys, but did not define a secure key exchange. Note that the format of the EAPoL-Key message in IEEE 802.11 is different from that in IEEE 802.1X-2001.

Figure 36 depicts the relationships among the Supplicant, associated with the STA, Authenticator, associated with the AP, and the Authentication Server (AS). While EAP messages are used between the Supplicant and AS, these messages are encapsulated in EAPoL frames as they are transmitted from Supplicant to Authenticator. Similarly, the EAP message may also be encapsulated over a “secure channel” between the Authenticator and AS. This secure channel is outside the scope of this specification. A typical implementation in practice, for example, might be based on Remote Authentication Dial-In User Service (RADIUS). Note that RADIUS is not mandated by the IEEE 802.11 or IEEE 802.1X standards, but RADIUS is a convenient protocol that may be used for this purpose. Like EAPoL, RADIUS has messages to augment EAP, for example, RADIUS may be used to transmit the pairwise master key (PMK) from the Authentication Server to the Authenticator, over the secure channel being provided by RADIUS or a protocol with similar attributes. The transmission of the PMK to the Authenticator is not accomplished using EAP messages, since EAP is an end-to-end protocol between the Supplicant and the AS.

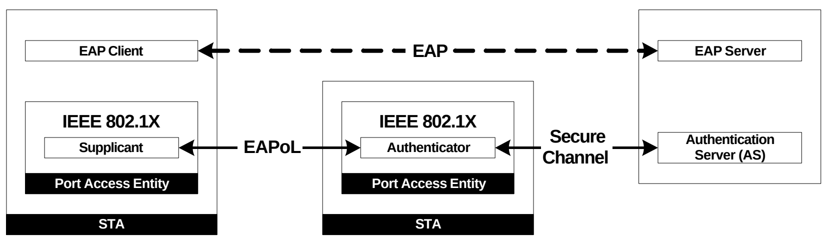


Figure 36—Authentication and key management overview

The EAP is not tied to any particular authentication algorithm, hence its extensibility. It defines a small number of messages used to communicate between the AS and the EAP Client. This design allows the two peer entities to mutually determine whether or not the newly connected device should be granted access to the network, based on the algorithm-specific authentication credentials, such as the user’s identification and password. The Authenticator is able to interpret the outcome of the negotiation without being required to participate in the negotiation itself, by simply recognizing an EAP-Success or EAP-Failure message.

EAPoL carries EAP messages between the Supplicant and the Authenticator. The Authenticator acts as a relay for EAP packets by extracting them from within the EAPoL frames and sending those EAP packets to the Authentication Server over the secure channel.

All EAPoL frames are normal IEEE 802.11 data frames, thus they follow the format of IEEE 802.11 MSDUs and MPDUs. With reference to the IEEE 802.11 frame format defined in clause 7.1.2, an MPDU may be up to 2346 octets in length, which encapsulates an MSDU payload that is up to 2312 octets in length. The remaining 34 octets in the MPDU comprise the IEEE 802.11 header (30 octets) and the four-octet Frame Check Sequence that concludes the frame.

EAPoL messages are just like any other data packet (MSDU) that might be transmitted over an IEEE 802.11 LAN, and as such are de-multiplexed using information contained in the LLC/SNAP header, which comprises the first eight octets after the MPDU header. The following figure illustrates an MPDU that contains an EAP packet, encapsulated in an EAPoL (IEEE 802.1X) header.

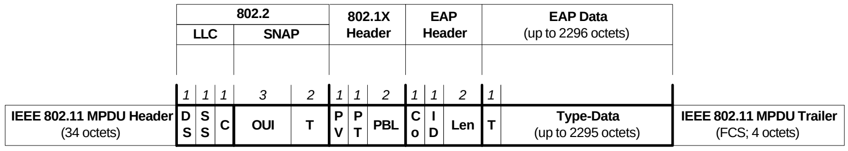


Figure 37—Full 802.11 MPDU format

The IEEE 802.2 LLC header's DS (Destination Service Access Point, or DSAP) and SS (Source Service Access Point, or SSAP) fields are both set to a value of 0xAA, which indicates that an IEEE 802.2 Sub-Network Access Protocol (SNAP) header follows the LLC header. The IEEE 802.2 LLC header's Control field is set to 0x03, indicating that this is an unnumbered information frame. To indicate that a standard Ethernet type is being used in the IEEE 802.2 SNAP header's Type field, the IEEE 802.2 SNAP OUI field is set to a value of 0x000000. A value of 0x888E in the SNAP header's Type field indicates that an IEEE 802.1X frame header is next.

The IEEE 802.1X header begins after SNAP's Type field, starting with the IEEE 802.1X Protocol Version (PV) field, the value of which is defined in the current IEEE 802.1X specification.. The next field is the one-octet IEEE 802.1X Packet Type (PT), which can take one of the five values, whose meanings are described in the following table.

0x00	EAP-Packet	Indicates that an EAPoL frame contains an EAP packet
0x01	EAPoL-Start	Used to initiate EAP protocol processing
0x02	EAPoL-Logoff	Not recommended for use with IEEE 802.11
0x03	EAPoL-Key	Used by the Authenticator and Supplicant to derive or exchange cryptographic keying information
0x04	EAPoL-Encapsulated-ASF-Alert	Used by a Supplicant to send ASF alerts prior to being fully authenticated

The IEEE 802.1X Packet Body Length (PBL) follows the Packet Type. Because the LLC/SNAP header is eight octets long, and the IEEE 802.1X header is an additional four octets, consuming a total of 12 octets of the MSDU, the IEEE 802.1X Packet Body Length (PBL) value can be at most 2300 octets (since the MSDU can be at most 2312 octets). The limit of 2300 is for unencrypted EAPoL-KEY messages. Note that in cases where the EAPoL-Key message is encrypted—using WEP, CCMP, TKIP, or WRAP—additional octets will be consumed which will effectively reduce the maximum MPDU payload capacity, hence the maximum PBL will not be able to be as large.

When the Packet Type field in an EAPoL packet is set to a value of 0x00 (meaning EAP-Packet), an EAP packet header follows the IEEE 802.1X header. The EAP packet header begins with a one-octet Code field that defines the function of the EAP packet. The EAP packet format is as follows:

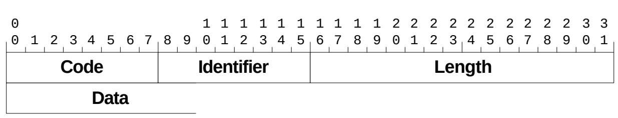


Figure 38—802.2 format

There are four EAP Codes: 0x01 (Request), 0x02 (Response), 0x03 (Success), and 0x04 (Failure). For EAP-Request or -Response packets, the one-octet Identifier field contains a value that is used to match Responses to Requests.

An EAP packet need not have a Data field, but a Data field will be present if the Code is set to Request or Response. For such EAP-Request and EAP-Response packets, the first octet of the Data field is a Type field that indicates which authentication algorithm is in use (e.g., EAP-TLS, PEAP, TTLS, etc.). The remainder of the Data field will be algorithm-specific data.

The STA initiates the association process. Once the STA and AP associate, the AP and STA will indicate success via one of the following APIs:

- MLME-ASSOCIATE.indication,
- MLME-ASSOCIATE.confirm,
- MLME-REASSOCIATE.indication, or
- MLME-REASSOCIATE.confirm.

If the AP is RSN-capable and configured with RSN is enabled, the EAPoL-Start message is sent by the AP, which is triggered once the STA and the AP complete their association. The completion of the association is detected by one of the APIs above. The AP advertises its RSN capabilities in its own configuration-dependent RSN IE. The AP constructs the IE based on the subset of its RSN capabilities enabled, and the AP then includes the RSN IE in its Beacon and Probe Response frames. The Supplicant's STA also constructs an RSN Information Element (RSN IE) that represents its configured RSN capabilities in the management frames that are used to facilitate association, which lets the Authenticator's STA (in the AP) know that this particular STA desires to join the RSN.

After the association first forms, only IEEE 802.1X protocol messages (i.e., EAP and its associated authentication method) flow across the link until authentication completes; the Supplicant's IEEE 802.1X Port Access Entity (PAE) filters all non-EAP traffic during this period. Until authentication completes with the distribution of a Pairwise Master Key (PMK), the PAE ensures that only EAP packets are sent or received between this STA and the wireless medium.

The authentication process allows the Authenticator and the Supplicant to prove to each other that they both know the PMK and it is essential that this be done without divulging the PMK to eavesdroppers. Even though the EAP Supplicant has been successfully authenticated by the Authentication Server, it cannot use the link until it has successfully derived the necessary encryption and authentication keys, which depend on the cipher suite chosen in the RSN IE in the AP's Beacon and Probe Response frames. The format of the RSN IE is as follows:

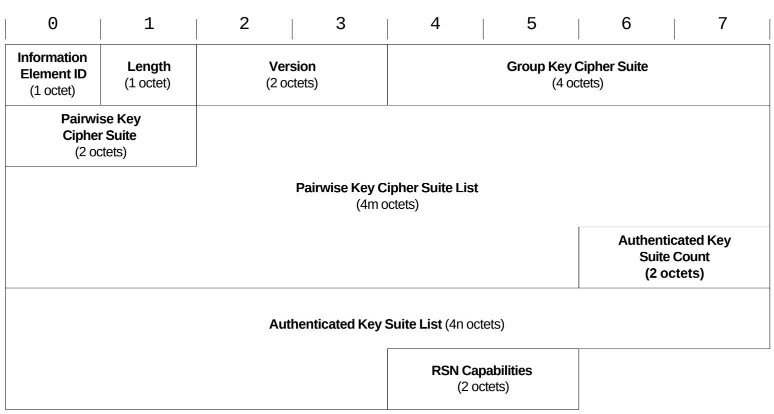


Figure 39—RSN Information Element

An EAP authentication method is negotiated as follows. One peer proposes an EAP authentication method to the other by sending an EAP-Request packet with the Type field's value set to the assigned number of the desired authentication method. If the receiving peer supports that authentication method, it will respond with an EAP-Response using the same Type as was proposed by the first peer. If the receiving peer does not support this authentication method, its EAP-Response packet will have the Type set to "NAK", and the original peer may then attempt to authenticate using a different method by proposing a different Type. A successful EAP authentication message flow is documented in the following figure.

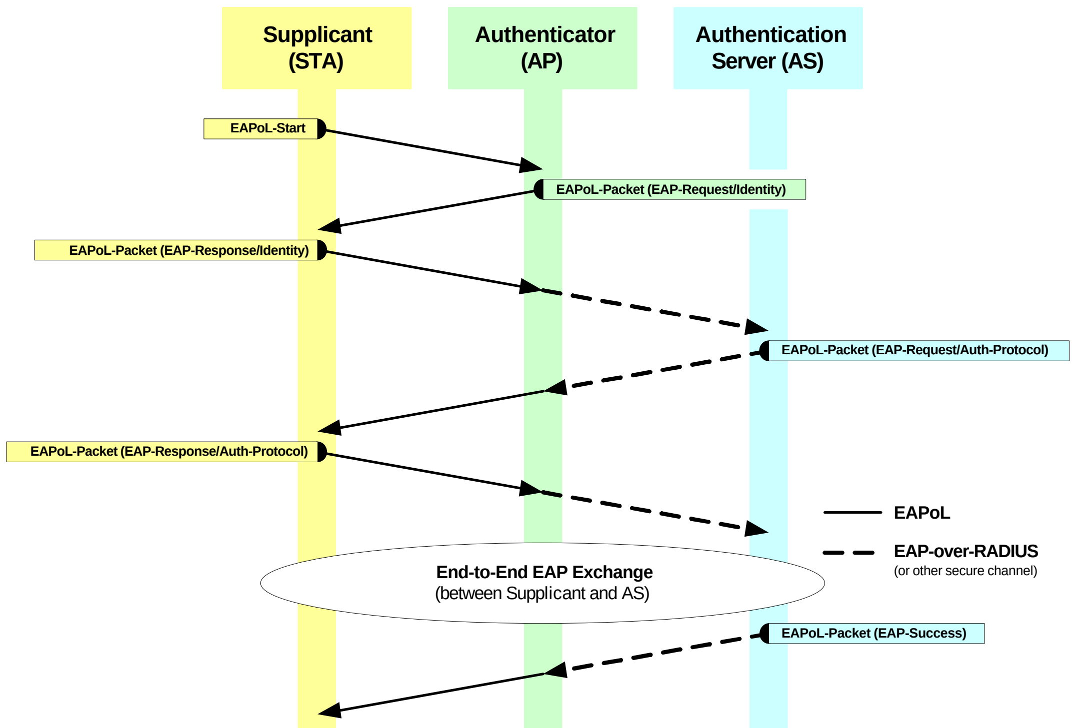


Figure 40—IEEE 802.1X authentication exchange

At the completion of a successful EAP authentication exchange, the AS informs the EAP Supplicant that the authentication has succeeded by sending an EAP-Success packet (Code = 0x03). The Authenticator is able to detect the EAP-Success code, and registers the fact that this EAP Supplicant now represents an authenticated station. Using the secure channel between the AS and the Authenticator, the AS also sends one other essential piece of information to the Authenticator, the Pairwise Master Key (PMK) that has been generated by both the EAP Supplicant and the AS. By virtue of the EAP Supplicant's authentication exchange with the AS, the EAP Supplicant already knows the PMK.

The Supplicant and the Authenticator cannot trust each other until they have securely determined that each party knows the PMK. In order to establish that trust relationship, the Authenticator and Supplicant use a "four-way handshake" to convince each other that they are who they claim to be, and to mutually derive the necessary encryption and authentication keys from the PMK. The four-way handshake does not reveal any essential keying information to eavesdroppers, but does provide each party with proof that they both know the PMK.

The following diagram depicts the four-way handshake, composed of EAPoL-Key messages. The parenthetical items next to each message are the "interesting" parts of each EAPoL-Key Descriptor. There are always nine elements in the EAPoL-Key Descriptor, but not all are relevant to each message:

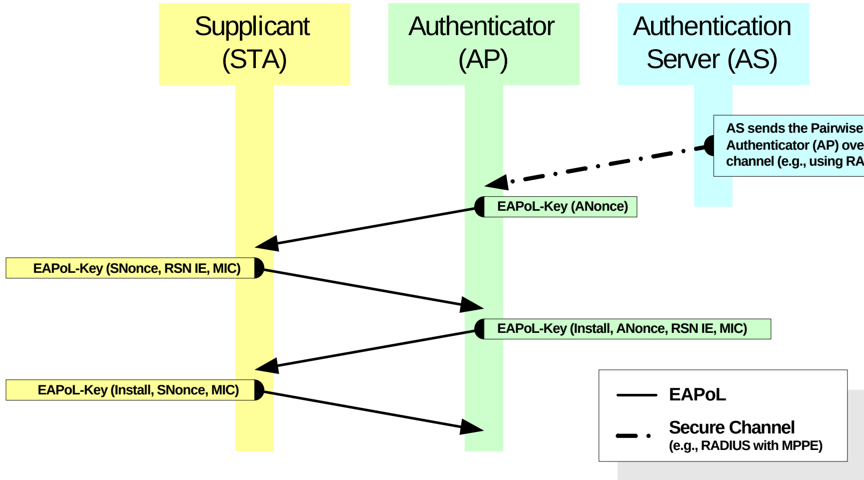


Figure 41—4-Way Handshake

A replay counter is part of each EAPoL-Key message, and enables detection (and thus prevention) of replay attacks. The replay counter is incremented by 1 for each successive message in the four-way handshake. Each retransmission of a given message uses the same replay counter value as was used when the message was first transmitted.

The first EAPoL-Key message of the four-way handshake is sent from the Authenticator to the Supplicant. The main purpose of the first message is to carry the randomly generated Authenticator Nonce (ANonce). Any observer could eavesdrop on this message and learn the Authenticator's chosen ANonce. Upon receiving the first message, the Supplicant has learned the ANonce. Subsequent messages in the four-way handshake ensure that only the legitimate Authenticator is in communication with the Supplicant.

Once the Supplicant has received the first message and generated its own SNonce, it has sufficient information to generate keys used for directed packet transmission and reception. Also, derived keys protecting (i.e., providing message integrity and confidentiality to) the remainder of the key exchange are derived from the ANonce contained in this first message, as well as the SNonce and the STA's RSN IE.

Any eavesdropper could also have attempted to impersonate the Authenticator by forging an EAPoL-Key message after it saw the EAP-Success packet. However, such an impostor would not know the PMK, thus it will not be able to successfully forge future EAPoL-Key messages, so the only exposure at this point is possibly to denial-of-service attacks.

The second EAPoL-Key message is from the Supplicant to the Authenticator, which acknowledges receipt of the first message. The second message contains a payload known as the RSN Information Element (RSN IE) that the Supplicant's STA has constructed based on the cipher suites it supports, and is the same RSN IE that the STA used during the association process. The AP has created its own RSN IE that defines which cipher suites are allowed to be used within this ESS. By sending its RSN IE to the Authenticator, the Supplicant informs the Authenticator of which cipher suites it supports, which controls how the keys are derived. Of the set of cipher suites that are supported by the STA and the set that is supported by the AP, a valid cipher suite is chosen from the intersection of those two sets.

The second message of the four-way handshake also transmits the Supplicant's Nonce (SNonce) to the Authenticator. Once the Supplicant has randomly generated its SNonce, it now has sufficient information to derive the necessary encryption and authentication keys that will be used during this security association, pending successful completion of the four-way handshake.

Finally, the second message also contains a digital signature that protects (i.e., is computed over) the entire EAPoL-Key packet, using one of the keys that the Supplicant has derived from the PMK and the two

Nonces, among other inputs. This digital signature is included in the second message in the MIC field of the EAPoL-Key Descriptor. The Authenticator will be able to verify this digital signature once it has received the second message from the Supplicant, and has itself derived the key that was used to compute this MIC field value. Only a Supplicant that knew both of the nonces and the PMK could have sent this message, since it contains a digital signature that could only have been computed if the PMK were known.

Like the first message, the second message is also sent in the clear (but as noted above, it is protected by the digital signature that is computed over the EAPoL-Key message and included in the EAPoL-Key Descriptor). The second message can also be observed by third parties, who also could have seen the ANonce and SNonce in the first and second message, as well as the Supplicant’s RSN IE, but who nonetheless cannot forge the digital signature (MIC) in the EAPoL-Key message without knowledge of the PMK.

The key derivation process alluded to above, in both the Supplicant and the Authenticator, is known as the “Pairwise Key Hierarchy”. The Pairwise Key Hierarchy defines how to combine the ANonce, the SNonce, the Authenticator’s MAC address (AA), the Supplicant’s MAC address (SA), and a specific ASCII string, as well as the PMK, as input to a pseudo-random function (PRF). The PRF outputs a large number of bits sufficient to define the EAPoL-Key encryption and message integrity check keys and the pairwise temporal key(s) for protecting unicast data traffic (the temporal keys are used for authentication and encryption). The length of the output of the PRF depends on the cipher suite that was determined based on comparing the RSN IEs in the association process.

Specifically, the PRF output is separated into the following components: the EAPoL-Key MIC Key (abbreviated MK; used to digitally sign the EAPoL-Key message), the EAPoL-Key Encryption Key (abbreviated EK; used to encrypt the EAPoL-Key Descriptor’s Key Material field during the Group Key Exchange, but it is not used in the four-way handshake that implements the pairwise key exchange; the EK is used to encrypt the EAPoL-Key Key Material field of the EAPoL-Key Descriptor in the Group Key Exchange), and the temporal key(s) for the cipher suite defined in the RSN IE. The Pairwise Key Hierarchy is illustrated in the following figure.

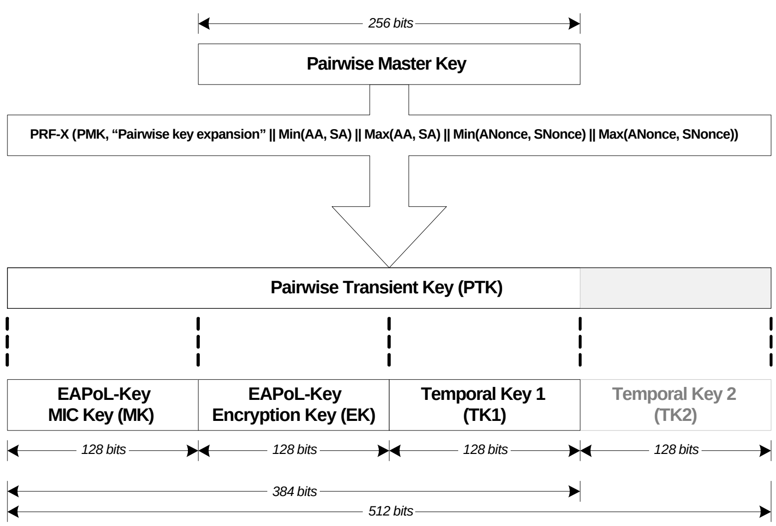


Figure 42—Pairwise key hierarchy

The complete output of the pseudo-random function (PRF) is known as the Pairwise Transient Key (PTK), of which bits 0 – 127 are the MK, bits 128 – 255 are the EK, and bits 256 – 383 represent temporal key number 1 (TK1). Temporal key number 2 (TK2), if present (which depends on the needs of the cipher suite defined in the RSN IE), is found in bits 384 – 511.

Note that the Authenticator cannot perform the PTK derivation until it has received the SNonce from the Supplicant, since the SNonce is part of the input in the PTK derivation. In other words, the Authenticator cannot execute the

Pairwise Key Hierarchy until after it has received the second message of the four-way handshake. The Authenticator and the Supplicant both derive identical temporal keys because they both compute the Pairwise Key Hierarchy using the same inputs. Because only this Supplicant and this Authenticator (and the Authentication Server) are presumed to know the PMK, no eavesdropper can learn enough information from simply observing the four-way handshake to impersonate the Supplicant or the Authenticator.

The third EAPoL-Key message of the four-way handshake is sent by the Authenticator to the Supplicant, and it is used to direct the Supplicant to install the temporal encryption key(s) in the Supplicant's STA. The third message sets the "Install" bit for the first time in the four-way handshake, as well as the ANonce (the same randomly chosen value that was sent in the first message), the RSN IE (must be identical to the RSN IE that was sent in the AP's Beacons and/or Probe Responses), and a digital signature computed over the third message's EAPoL-Key packet by the Authenticator using the MK that it has now derived.

When set, the EAPoL-Key message's Install bit directs the receiver to configure its local STA with the derived temporal key(s). In the case of the third message the Supplicant is the receiver of the message, so the Authenticator is using the Install bit to tell the Supplicant to prepare to receive encrypted unicast traffic. The third message is similar to the first message, but it conveys much more information, built on what has been learned in the first and second messages.

The final EAPoL-Key message of the four-way handshake is very similar to the second message. In this message, the Supplicant is directing the Authenticator to install the per-association temporal key(s) into the Authenticator's STA. The fourth message is stating that the Supplicant has installed the temporal encryption key(s) in its STA and is ready to receive unicast data encrypted using the cipher suite specified in the RSN IE. As with the second and third messages, the fourth message contains a digital signature that is computed over the EAPoL-Key message using the MK. Since the fourth message acknowledges the third message, it tells the Authenticator that the temporal keys have been installed on the Supplicant's STA. Furthermore, by virtue of the Install bit being set in the fourth message, the Supplicant is directing the Authenticator to install the temporal keys for this security association into its STA (i.e., in the AP). The entire fourth message is encrypted using the temporal keys and the cipher suite that has been negotiated prior to this point in the four-way handshake.

Once the keys have been installed, the AP's STA can send encrypted unicast traffic to the Supplicant's STA. The fourth message's EAPoL-Key Descriptor contains a digital signature over the EAPoL-Key message, which is digitally signed (i.e., MIC'ed) using the MK. This MIC field was computed as in the second and third messages. In contrast to the previous messages, the fourth message is not sent in the clear, but is encrypted using the derived temporal key(s) using whatever unicast cipher suite was defined in the RSN IE. Thus, the fourth message will be encrypted using CCMP, TKIP, or WRAP.

If the fourth message does not reach the Authenticator, the Supplicant's STA must still be prepared to accept unencrypted traffic from the Authenticator (which would most probably be a re-transmission of the third message, since the Authenticator will not have received the fourth message from the Supplicant, which, among other functions, serves to acknowledge the third message from the Authenticator). Provided the fourth message has been properly received and interpreted by the Authenticator, the per-association keys are installed on the Authenticator's STA, and future unicast data is encrypted using TK1 and/or TK2, as required by the RSN IE. Once the four-way handshake is complete, the Authenticator's and Supplicant's IEEE 802.1X PAE permits unicast traffic to flow through their respective STAs, which encapsulates the packets according to the cipher suite(s) indicated in the RSN IE.

The "Install" bit in the third and fourth messages directs the IEEE 802.1X entity in the Supplicant or the Authenticator, respectively, to configure its local STA with the keying information derived from the PTK. The API that is used to convey this information from the 802.1X entity to the STA is the MLME-SETKEYS.request. In the event that an Authenticator or Supplicant decides to terminate an association, the MLME-DELETEKEYS.request API is used.

Now that the unicast pairwise key hierarchy calculations have been completed, unicast traffic must be sent in encrypted form, using the derived temporal keys. However, multicast and broadcast traffic would still need to be sent in the clear, which is why there is a small additional handshake (two messages) in which the Authenticator transmits the Group Transient Key (GTK) to the Supplicant.

All the STAs in an ESS use the same Group Transient Key, but the Authenticator securely delivers it to each authenticated Supplicant, in a process that is protected by the unicast temporal encryption keys that have now been derived. The EAPoL-Key messages of the GTK exchange are encrypted using unicast key(s) derived from the PTK. The *encrypted* Group Key exchange is illustrated in the following diagram:

Figure 43—Group key handshake

The Group Key Hierarchy involves a similar calculation to the Pairwise Key Hierarchy, in which the Authenticator derives the Group Transient Key from the Group Master Key, the Authenticator's [MAC] Address (AA), and the GNonce, as shown in the following diagram:

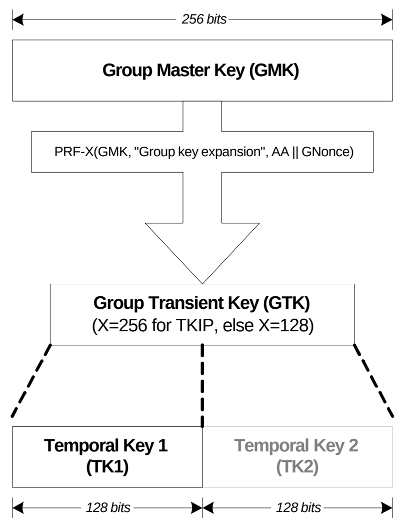


Figure 44—Group key hierarchy

As noted in the diagram, when TKIP is the cipher suite indicated in the RSN IE, the PRF is set to output 256 bits of GTK, so that the Group Temporal Key 2 will also be derived, which is the second 128 bits of the output of the PRF. Otherwise (e.g., in the cases of CCMP and WRAP), the GTK is only 128 bits long. In such cases, the PRF's output is just 128 bits long, and those 128 bits are directly mapped into the Group Temporal Key 1.

Based on the contents of the RSN IE (i.e., whether or not TKIP is in use), a Supplicant that receives the encrypted GTK from the Authenticator is able to decipher one or two Temporal Keys from the GTK that it receives from the Authenticator. Both of the EAPoL-Key messages in the Group Key Exchange are digitally signed by the MK, after the EK has been used to encrypt the Key Material field of the EAPoL-Key Descriptor, which holds the GTK. The Group TK1 (and possibly also TK2), are subsequently configured into the Supplicant's STA and the Authenticator's STA via the MLME-SETKEYS.request API. When this procedure is complete, the Supplicant's STA can now send encrypted broadcast and multicast traffic, in addition to the prior ability to send encrypted unicast traffic.

8.4.2 RSN selection

In an RSN or a TSN STAs (including APs) shall advertise their capabilities by asserting the Robust Security bit of the Capabilities Information Field in Beacon and Probe Response messages. In an RSN a STA may also include the RSN Information Element (RSN IE, see 7.3.2.17) in Beacons, and Probe Responses. When doing so, the included RSN IE shall specify all the authentication and cipher suites enabled by its policy. An RSN-capable STA operating as part of a TSN may omit the RSN IE from its Beacons and Probe Responses. A STA shall not advertise any authentication or cipher suite that is not enabled and that it will not agree to use.

The STA's IEEE 802.11 Management Entity shall utilize the MLME-SCAN.request to identify neighboring STAs that assert Robust Security and advertise an SSID identifying an authorized ESS or IBSS. A STA may decline to communicate with STAs that do not assert Robust Security, or do not advertise an authorized SSID. A STA may also decline to communicate with other STAs that do not advertise authorized authentication and cipher suites with its RSN IE.

A STA shall advertise the same RNSE in both its Beacons and Probe Responses.

Informative Note: Whether or not a STA may attempt to communicate with another STA that asserts Robust Security but which does not advertise an authorized SSID is a matter of policy.

Informative Note: Whether a STA with Robust Security enabled may attempt to communicate with a STA that does not assert RSN is a policy question.

Informative Note: It should be possible to independently enable or disable the following in an RSN AP:

- RSN
- TSN
- WEP using pre-RSN IEEE 802.1X key management
- WEP without key management.

For RSN an AP should support TKIP as well as CCMP.

Informative Note: It should be possible to independently enable or disable the following in an RSN STA:

- RSN
- WEP using pre-RSN IEEE 802.1X key management
- WEP without key management.

Informative Note: As a practical matter, the multicast cipher suite must be the weakest unicast cipher suite enabled.

Informative Note: An AP should support pre-shared keys.

In an IBSS a STA may also identify another STA as belonging to the same IBSS by receiving a protected message with A3 asserting the BSSID of the IBSS. If a STA does not already have a security association with the message source, the receiver will not have cryptographic keys to decapsulate messages it receives from that STA. On receiving a protected message from such a STA, the receiver should attempt to initiate a security association, as described in 8.4.1.

Informative Note: Typically this sort of message will be broadcast/multicast. It is also possible to receive a protected unicast message after a STA has reset in a way that is undetectable to the message source.

Similarly, if a STA in an IBSS receives the first message of a 4-way handshake from an unknown STA asserting the IBSS BSSID as A3, the STA's IEEE 802.1X implementation should respond, in an attempt to establish a security association.

8.4.3 RSN policy selection in an ESS

RSN policy selection in an ESS utilizes the normal IEEE 802.11 association procedure. RSN policy selection is performed by the associating STA. The STA does this by including an RSN IE in its (Re)Association Requests.

In an RSN an AP shall not associate with pre-RSN STAs, i.e., STAs that fail to assert RSN.

> Informative Note: This can be enforced by configuring the AP to use only RSN cipher and authentication suites, i.e., by disabling WEP and pre-RSN IEEE 802.1X key management.

The STA initiating an association shall insert an RSN IE into its (Re)Association Request whenever the targeted AP indicates RSN support. The initiating STA's RSN IE shall include one authentication and pairwise cipher suite from among those advertised by the targeted AP in its Beacons and Probe Responses. It shall also specify the group key cipher suite specified by the targeted AP. If at least one RSN IE field from the AP's RSN IE fails to overlap with any value the STA supports, the STA shall decline to associate with that AP. It is invalid in an RSN to specify "None" as the Pairwise cipher.

If an RSN-capable AP receives a (Re)Association Request including an RSN IE, and if it chooses to accept the association, the AP shall, to secure this association use the authentication and pairwise key cipher suites the RSN IE in the (Re)Association Request specifies.

A STA shall observe the following rules when processing an RSN IE:

- A STA shall advertise the highest Version it supports.
- A STA shall request the highest Version field value it supports among all those a peer STA advertises.
- STAs without overlapping supported Version field values shall not use RSN methods to secure their communication.
- A STA shall ignore OUI values it does not recognize.

In order to accommodate local security policy, a STA may choose not to associate with an AP that does not support any pairwise key cipher suite.

8.4.3.1 TSN policy selection

If the AP includes the RSN IE in its Beacons or Probe Response messages, the forgoing applies in a TSN—RSN STAs shall act as if it is operating in an RSN, by including the RSN IE in its (Re)association requests. A STA may omit the RSN IE from (Re)association Requests it transmits to APs that fail to include the RSN IE in their Beacon and Probe Response messages, and the STA shall not use RSN methods with such an AP; instead, it shall use a pre-configured WEP key to secure its communication.

An RSN-capable AP configured to operate in a TSN may include the RSN IE, and shall associate with both RSN and pre-RSN STAs. This means that an RSN-capable AP shall respond to an associating STA that includes the RSN IE just as in an RSN.

If an AP operating within a TSN receives a (Re)association request without an RSN IE, it shall allow communications only if a WEP key has been configured to secure communication. If a WEP key is not installed, the AP shall reject the association request; if a WEP key is configured, the AP may accept the request.

An AP cannot support multiple group key cipher suites simultaneously within an ESS. In particular, a TSN must use the cipher suite supported by the least capable STA it admits as the group key cipher suite.

8.4.4 RSN policy selection in an IBSS

The IEEE 802.1X implementations of two directly communicating STAs negotiate pairwise key cipher suites using the 4-way handshake. Thus, each pair of STAs within an IBSS may use IEEE 802.1X to negotiate its own pairwise key cipher suite. As specified in 8.5.2, Messages 2 and 3 of the 4-way handshake convey an RSN IE. The Message 2 RSN IE includes a list of allowed pairwise key cipher suites, and the RSN IE in Message 3 reports the selected the pairwise key cipher suite; the Message 3 RSN IE shall specify a pairwise key cipher suite from those suggested in Message 2, or else the 4-way handshake shall fail. Beacons and Probe Responses within an IBSS shall specify an empty list of pairwise key cipher suites.

> Informative Note. An IBSS does not use the Beacon/Probe Response negotiation mechanism, as knowledge of a peer STA within an IBSS may not come from the Beacon or Probe Response source.

The IEEE 802.1X implementations shall check that the group key cipher suite and authenticated key management protocol match those in the Beacons and Probe Responses for the IBSS. IEEE 802.1X can extract this information from IEEE 802.11.

> Informative Note: The RSN information elements in message 2 and 3 are not the same as in the MAC messages, the multicast cipher and AKMP are the same but the unicast ciphers may be different.

> Informative Note: When an IBSS network uses pre-shared keys, STAs can negotiate a unicast cipher. However, any STA in the IBSS can derive the pairwise keys of any other that uses the same pre-shared key by capturing the first two messages of the 4-way handshake.

8.4.4.1 TSN policy selection

Non-RSN STAs generate Beacons and Probe Responses without an RSN IE, and will ignore the RSN IE, while RSN stations will include the RSN IE in Beacons and Probe Responses. This allows an RSN STA to identify the non-RSN STAs from which it has received Beacons and Probe Responses. If an RSN STA instead identifies another IBSS member on the basis of a received broadcast/multicast message, it cannot make this judgment directly.

If an RSN STA in a TSN IBSS cannot identify a newly identified peer as RSN, it may treat the new STA as non-RSN and attempt to communicate with it using WEP and a default WEP key.

8.4.5 MPDU filtering

When the policy selection process chooses IEEE 802.1X authentication, a STA (including AP) shall filter all non-IEEE 802.1X class 3 MPDUs after association completes but prior to the completion of IEEE 802.1X authentication and key management.

> Informative Note. Filtering class 3 MPDUs is not required during pre-authentication.

Explicitly, the STA shall begin this filtering when the MLME-ASSOCIATE.indication, MLME-ASSOCIATE.confirm, MLME-REASSOCIATE.indication, or MLME-ASSOCIATE.confirm indicates it has formed a new association with a peer STA.

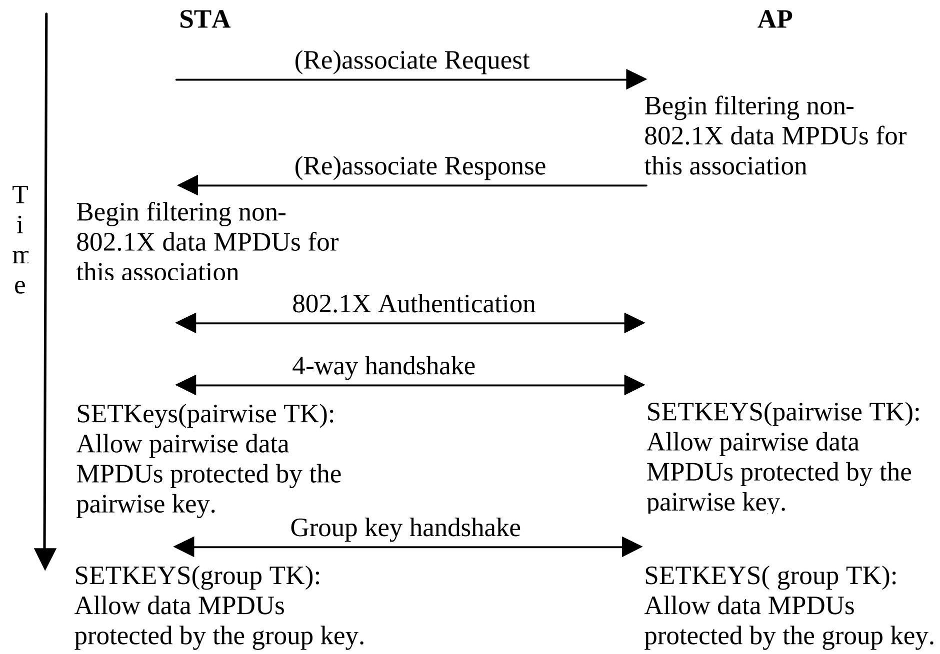


Figure 45—Sequence of Filtering-related Events

The STA shall relax this filtering to permit authorized unicast MPDUs when IEEE 802.1X uses the MLME-SETKEYs.request to initialize pairwise temporal keys for the association. The STA shall relax this filtering to permit authorized broadcast/multicast MPDUs when IEEE 802.1X uses the MLME-SETKEYS.request to initialize the group temporal key for the association.

By definition, authorized MPDUs shall be

1. received IEEE 802.1X messages.

   Informative Note. It is assumed that the IEEE 802.1X Supplicant or Authenticator will discard received IEEE 802.1X messages that are not relevant to the current state, e.g., ones not protected by the current pairwise master key.

2. received unicast class 3 MPDUs successfully protected by the agreed-upon temporal key;

3. received multicast/broadcast class 3 MPDUs successfully protected by the agreed upon temporal group key.

4. once a temporal key is configured, any class 3 MPDU to be transmitted as a unicast;

5. once a group temporal key is configured, any class 3 MPDU to be transmitted as a multicast or broadcast.

Informative Note: In a TSN the group key may be used for unicast communication as well as broadcast/multicast communication. In this case IEEE 802.1X does not configure the pairwise key.

Figure 45 depicts a time-sequence diagram of the events related to filtering.

8.4.6 RSN authentication in an ESS

When IEEE 802.1X authentication is an authentication option, an RSN-capable STA may use IEEE 802.11 Open System Authentication prior to association or Reassociation.

Informative Note: IEEE 802.1X authenticates in a layer above the IEEE 802.11 MAC. It removes authentication processing from the IEEE 802.11 MAC and delegates this function to IEEE 802.1X. A STA may become authenticated via IEEE 802.1X if *dot11AuthenticationType* at the recipient STA is set to IEEE 802.1X authentication. IEEE 802.1X authentication may fail, as a STA may decline to authenticate with any other STA.

IEEE 802.1X authentication is initiated by any one of the following mechanisms:

1. If a STA negotiates to use IEEE 802.1X authentication during (re)association, the STA's management entity can respond to the MLME-ASSOCIATE.confirm (resp. indication) by requesting the STA's Supplicant (resp. AP's Authenticator) to initiate IEEE 802.1X authentication. Thus, in this case, authentication is driven by the STA's decision to associate and the AP's decision to access the association.

2. If a STA's MLME-SCAN.indication finds another AP within the current ESS, a STA may signal its Supplicant to use IEEE 802.1X to pre-authenticate with that AP.

   Informative Note: The IEEE 802.1X Supplicant of a roaming STA initiates pre-authentication by sending an EAP-Start message to a new AP via its old AP and the DS.

3. If a STA receives an IEEE 802.1X message, it delivers this to its Supplicant or Authenticator, which may initiate a new IEEE 802.1X authentication.

   Informative Note: The IEEE 802.1X Authenticator of an AP initiate authentication by sending an EAP-Request/Identity message to the Supplicant of a STA.

   Informative Note: When a STA (re)associates with an AP without a (recent enough) pre-authentication, the AP has no cryptographic keys configured for the STA. In this case, the AP's Authenticator will force a full IEEE 802.1X authentication. In the case where the STA has recently pre-authenticated with the AP, the AP will retain the STA's IEEE 802.1X identity and cryptographic keys from the pre-authentication. In this case, the AP's Authenticator may proceed directly to key management in response to the STA's Supplicant's EAP-Response/Identity.

   Informative Note: Pre-authentication completes when the AP's IEEE 802.1X Authenticator sends the first message of the 4-way handshake to the STA's IEEE 802.1X Supplicant.

   Informative Note: If IEEE 802.1X authentication completes successfully, the AP's Authenticator forwards an EAP-Success message to the STA's Supplicant and then initiates the 4-way handshake, to complete key management. If IEEE 802.1X authentication fails, the AP's Authenticator uses the MLME-DEAUTHENTICATE.request primitive to inform IEEE 802.11 of the problem.

The AP shall respond to an IEEE 802.1X authentication failure by sending the STA a Disassociation message.

A STA (including an AP) shall pass IEEE 802.1X data frames. Being data frames, they shall be sent in the clear if no pairwise keys have been established by key management, and the established pairwise keys shall protect the IEEE 802.1X data frames otherwise.

Informative Note: There is a potential race condition with the final IEEE 802.1X message when an association begins, in that it may be sent unencrypted. Accordingly the filtering rules in 8.4.5 require the MAC to pass all IEEE 802.1X messages even if keys have been configured. This sort of race condition is inherent in all key management schemes, and cannot be removed by "clever" design.

If a STA is associated with an AP, it shall disassociate if IEEE 802.1X authentication with that AP's Authenticator fails. If IEEE 802.1X authentication fails, a non-AP STA may associate again with the same to reinitiate the process, or attempt to associate with another AP.

Informative Note: IEEE 802.1X uses the MLME-DEAUTHENTICATE.request primitive to inform the 802.11 MAC when authentication failed.

Informative Note: There is no requirement to disassociate with the associated AP if pre-authentication with a different AP fails.

8.4.6.1 Pre-authentication and key management (Informative)

A STA shall not use pre-authentication except when pairwise keys are employed.

When pre-authentication is used, then

1. Authentication is independent of roaming.
2. the STA's Supplicant may be authenticate with multiple APs at a time.

Informative Note. Pre-authentication can be useful as a performance enhancement, as Reassociation will not include the cost of a full reauthentication when it is used.

Pre-authentication relies on IEEE 802.1X. A STA can initiate pre-authentication whenever it has a link established with an AP. To effect pre-authentication, the STA sends an IEEE 802.1X EAP-Start message as a data frame to the BSSID of a targeted AP via the AP with which it is associated. Thus, the STA sets the To DS subfield in the Frame Control Field. It is the responsibility of the associated AP to forward the data frame to the targeted AP via the DS.

An AP's Authenticator that receives an EAP-Start message via the DS may initiate 802.1X authentication by sending an EAP-Request/Identity to the STA via the DS. The DS will be configured to forward this message to the AP with which the STA is associated. The pre-authentication exchange ends when the Authenticator sends the first message of the 4-way handshake.

A STA may initiate pre-authentication with any AP within its present ESS with pre-authentication enabled, whether or not the targeted AP is within radio range.

Informative Note: Pre-authentication is a MAC level mechanism, so cannot be used across, .e.g., IP subnet boundaries.

If pre-authentication is not used, the STA must make a roaming decision prior to authentication. Data transfer will halt during the IEEE 802.11 authentication and association, the IEEE 802.1X authentication, and IEEE 802.1X key management.

When pre-authentication is used, the STA's IEEE 802.1X Supplicant must cache the PMK for some period, in case the STA associates with the AP with which the STA's Supplicant has pre-authenticated.

Similarly, the AP's IEEE 802.1X Authenticator must cache the PMK key for some period in case the pre-authenticated STA associates with the AP. If during authentication the AP's Authenticator finds it has cached the PMK for the associated STA, it may respond with an immediate EAP-Success message and then initiate the 4-way handshake.

Both the Supplication and the Authenticator may delete a cached PMK if the pre-authenticated STA does not associate with the selected AP after some time interval.

Informative Note: Even if a STA has pre-authenticated, it is still possible that it may have to undergo a full IEEE 802.1X authentication, as the AP's Authenticator may have purged its PMK due to, e.g., unavailability of resources, or slowness of the STA to authenticate, etc.

Pre-authentication can fail, and an AP's Authenticator or STA's Supplicant can destroy keys established by pre-authentication prior to association. If the AP's Authenticator loses pre-authentication keys in this manner, it shall send an IEEE 802.11 Deauthentication message on receiving any encrypted packets from the station.

Pre-authentication introduces new opportunities for denial-of-service attack. To limit the efficacy of these attacks, STAs (including APs) shall rate-limit IEEE 802.1X messages. STAs shall ignore IEEE 802.1X from APs with which it is neither associated nor pre-authenticating.

8.4.7 RSN authentication in an IBSS

When authentication is used in an IBSS, it is driven by the STA wishing to establish communications. The Management Entity of this STA chooses a set of STAs with which it may want to authenticate, and then may cause the MAC to send an IEEE 802.11 Open System Authentication message to each targeted STA. Targeted STAs that wish to respond will return an IEEE 802.11 Open System Authentication message to the initiating STA. The STA Management Entity will then request its local IEEE 802.1X Supplicant to authenticate to the Authenticator of each responding STA. The STA's Supplicant begins the authentication process by sending an EAP-Start message to the Authenticator.

When it receives an MLME-Authentication.indicate due to an Open System Authentication Request, the IEEE 802.11 Management Entity on a targeted STA shall respond with an Open System Authentication Response and then request its Authenticator to begin IEEE 802.1X authentication, i.e., to send an EAP-Request/Identity message to the Supplicant.

The IEEE 802.1X messages are sent as IEEE 802.11 data messages. The data messages are sent with the FromDS and ToDS bits set to 0 and they are sent unencrypted since no keys are available.

The EAPOL-Key message is used to exchange information between the Supplicant and the Authenticator to negotiate a fresh pairwise temporal key. There is a single Pairwise key between the Supplicant and Authenticator produced by the 4-way handshake. The Pairwise key is used to transfer Group key updates and may be used as a Pairwise transient key.

8.4.8 RSN key management in an ESS (Informative)

When the IEEE 802.1X authentication per se completes, the STA's IEEE 802.1X Supplicant and the IEEE 802.1X AS will share a secret, called a *Pairwise Master Key* (*PMK*). The PMK acts as a master session key. The final step of security association set up occurs when the AS transfers the PMK to the AP with which the STA is associated, followed by a key confirmation handshake between the STA and the AP. The key confirmation handshake effectively replaces the function played by the IEEE 802.1X Success message in a secure wired network.

The key confirmation handshake is effected by an IEEE 802.1X protocol called the *4-way handshake*. The purposes of the 4-way handshake are

1. to confirm the existence of the PMK at the peer;
2. to insure that the security association keys are fresh, and
3. to synchronize the installation of session keys into the MAC.

The first message of the 4-way handshake is also utilized to signal the successful completion of a pre-authentication exchange.

The 4-way handshake is implemented using EAPOL-Key messages, described in 8.5.

> Informative Note. Neither the AP nor the STA can use the PMK for any purpose but the one specified herein without compromising the key. If the AP uses it for another purpose, then the STA can masquerade as the AP; similarly if the STA reuses the PMK in another context, then the AP can masquerade as the STA. These problems are possible because the IEEE 802.1X architecture as currently formulated does not explicitly bind the PMK to this particular session between the AP and the STA.

IEEE 802.1X signals the completion of key management by utilizing the MLME-SETKEYS.request to configure the agreed-upon temporal pairwise key into the 802.11 MAC.

A second key exchange is also defined, to distribute a temporal group key. This is called the ***group key handshake***. When the 4-way handshake completes, the AP's Authenticator can use the group key handshake to transfer the temporal group key for the Group Key cipher suite to the STA's Supplicant, to allow the STA to receive "secure" broadcast/multicast traffic. The group key handshake uses the EAPOL-Key messages for this exchange. When it completes, the STA can use the MLME-SETKEYS.request primitive to configure the temporal group key into the IEEE 802.11 MAC.

The AP may queue a Group key update message it cannot immediately send. If the AP later deletes this message prior to its transmission, the AP should disassociate the STA.

8.4.9 RSN key management in an IBSS

To establish a security association between two STAs in an IBSS, each STA shall support an IEEE 802.1X Authenticator and Supplicant, and each Authenticator initiates the 4-way handshake with the other STA's Supplicant.

The 4-way handshake is used to negotiate the pairwise key cipher suites. This is accomplished by include an RSN IE in the exchange initiated by the Authenticator whose STA has the lower MAC address. Message 2 of this exchange contains a list of pairwise key cipher suites, and Message 3 contains a single unicast cipher. If this exchange negotiates a pairwise key cipher suite, IEEE 802.1X installs the temporal key portion of the Pairwise Transient Key into the IEEE 802.11 MAC. Each Authenticator also uses the PTK negotiated by the exchange it initiates to distribute its own Group Transient Key. Each Authenticator generates its own Group keys, and uses the Group Key handshake to transfer the GTK to other STAs with whom it has completed a 4-way handshake.

A STA's IEEE 802.1X implementation shall check that the multicast cipher and AKMP matches that in Beacons and Probe Response received for the IBSS.

8.4.10 RSN security association termination

When a STA disassociates or deauthenticates, it shall delete any pairwise or group keys configured. Similarly, if a non-AP STA receives the MLME-ASSOCIATE.request or MLME-REASSOCIATE.request primitive when pairwise or group keys are configured, it shall delete them, If an AP receives a (Re)Association Request message from a STA that is already associated, it shall delete any pairwise keys associated with that STA.

8.4.10.1 Disassociate and Deauthentication message handling

Since key management is independent of the IEEE 802.11 state, keys may or may not be available in each of these states, so Deauthentication and Disassociate messages may or may not be sent when keys are available.

There are a number of abnormal situations that can cause a STA or AP to lose state. For example, a STA may be in State 3 when its associated AP is in State 1. The STA will protect data messages it sends to the AP. Then the AP cannot decapsulate messages it receives from the STA. The AP needs to send a Deauthentication message to the STA to force it into State 1.

Under normal circumstances STAs do not send Disassociate or Deauthentication messages, because the roam out of range or their user powers them off. Instead, APs commonly use a timeout to remove association state. A common case occurs when a STA, wanting to form a new association, is in State 1 and the AP is in State 3, timing out a prior association. This action needs to clear the AP's association state for this STA.

The following cases occur:

1. The AP needs to accept authenticate messages without being able to validate them, to handle STAs moving out of range.

2. The AP needs to accept associate messages without being able to validate them, to handle the first time associate.

3. A STA needs to accept Deauthentication messages without being able to validate them, to handle an AP restarting or otherwise losing the STA's association. APs also time out association state when no traffic is received from the STA.

The APs response to Disassociate and Deauthentication messages are in the following table:

Table 4—AP response to Disassociate and Deauthentication messages

AP state	**IEEE 802.1X portSecure**	**AP response to Disassociate or Deauthentication messages**	**AP response to other messages**
1	N	Process message	Process message
1	Y	Process message	Process message
2	N	Process message	Process message
2	Y	IEEE 802.1X indicate to MLME	Process message
3	N	Process message	Process message
3	Y	IEEE 802.1X indicate to MLME	Process message

This changes the handling of received Deauthentication and Disassociate messages when keys are available. This does not affect the procedures for the MLME-Deauthentication and MLME-Disassociate interfaces. In the received message case, an IEEE 802.1X re-authentication is requested. Failure of the IEEE 802.1X authentication returns the AP to State 1, generating a Deauthentication message by calling the MLME-Deauthenticate.Request interface.

The MLME SAP interface shall still indicate disassociate or Deauthentication indications but the MLME should not change the STA state. The MLME may initiate an IEEE 802.1X re-authentication depending on its knowledge of the IEEE 802.1X authentication state.

Table 5—non-AP STA response to Disassociate and Deauthentication messages

STA state	802.1X portSecure	STA response to Disassociate or Deauthentication messages	STA response to other messages
1	N	Process message	Process message
1	Y	Process message	Process message
2	N	Process message	Process message
2	Y	IEEE 802.1X indicate to MLME	Process Message
3	N	Process message	Process message
3	Y	IEEE 802.1X indicate to MLME	Process message

This changes the handling of receiving Deauthentication and Disassociate messages when keys are available. In this case, an IEEE 802.1X re-authentication is requested. If IEEE 802.1X authentication fails, this returns the STA to State 1 and causes it to send a Deauthentication message.

The MLME SAP interface shall still indicate disassociate or Deauthentication indications, but the MLME should not change the STA state. The MLME may initiate an IEEE 802.1X re-authentication depending on its knowledge of the IEEE 802.1X authentication state.

8.4.10.2 Illegal data transfer

In an RSN a STA and an AP transfer only protected data packets, with the only unprotected data packets allowed being unicast IEEE 802.1X message; these are permitted only when no Pairwise key is shared between the STA and the AP. If the STA and AP key state gets out of synchronization the following rules apply:

1. If an AP receives a unicast protected packet when it does not have keys to decapsulate, it shall send a Disassociate message to the STA and discard the data packet.

2. If a non-AP STA receives a unicast protected packet when it does not have keys to decapsulate the packet, it shall discard the data packet and send a Disassociate message to the AP; if the STA wants communications to continue, it should follow the Disassociate message with an immediate associate request to the AP.

3. On receiving a Disassociate or Deauthentication message, a STA shall delete the Pairwise key and, if it wants to continue communications, Reassociate to an AP of the same ESS.

8.5 Keys and key distribution

8.5.1 Key hierarchy

RSN defines two key hierarchies:

1. Pairwise key hierarchy, to protect unicast traffic; and

2. Group key hierarchy, to protect multicast traffic.

Informative Note: Pairwise key support with TKIP, WRAP, or CCMP allows a receiving STA to detect MAC address spoofing and data forgery. The RSN architecture binds the transmit and receive addresses to the pairwise key. If an attacker creates an MPDU with the TA, then the decapsulation procedure at the receiver will generate an error. Group keys do not have this property.

The description of the key hierarchies uses the following two functions:

- L (*Str*, *F*, *L*) From *Str* starting from the left, extract bits *F* through *F*+*L* bits, using the 802.11 bit conventions from 7.1.1.
- PRF-*n* Pseudo-random function producing *n* bits of output, defined in 8.5.1.

The symbol *AA* denotes the IEEE 802.1X Authenticator MAC Address, and *SA* denotes the Supplicant's MAC Address. In an ESS, *AA* is the wireless MAC address of the AP, and *SA* the MAC address of the STA.

A STA shall support a single pairwise key for any TA/RA pair. The TA/RA identifies the pairwise key, which does not correspond to any WEP key id. Group keys shall not use WEP key id 0. Instead, a group key is identified by WEP key id 1 or 2 and the TA/RA pair.

8.5.1.1 PRF

A Pseudo-Random Function (PRF) is used in a number of places in this document. Depending on its use it may need to output 128 bits, 192 bits, 256 bits, 384 bits or 512 bits. This section defines five functions:

- PRF-128, which outputs 128 bits,
- PRF-192, which outputs 192 bits,
- PRF-256, which outputs 256 bits,
- PRF-384, which outputs 384 bits, and
- PRF-512 which outputs 512 bits.

In the following, *A* is a unique label for each different purpose of the PRF; *Y* is a single octet containing 0, *X* is a single octet containing the parameter, and || denotes concatenation as usual.

H-SHA-1(K, A, B, X) $\leftarrow$ HMAC-SHA-1(K, $A \,\|\, Y \,\|\, B \,\|\, X$)
PRF-128(K, A, B) = PRF(K, A, B, 128)
PRF-192(K, A, B) = PRF(K, A, B, 192)
PRF-256(K, A, B) = PRF(K, A, B, 256)
PRF-384(K, A, B) = PRF(K, A, B, 384)
PRF-512(K, A, B) = PRF(K, A, B, 512)

```
PRF(K, A, B, Len)
    for i ← 0 to (Len+159)/160 do
        R ← R || H-SHA-1(K, A, B, i)
    return L(R, 0, Len)
```

8.5.1.2 Pairwise key hierarchy

The Pairwise key hierarchy utilizes PRF-384 or PRF-512 to derive session specific session keys from a PMK, as depicted in Figure 46. The PMK shall be 256 bits. The Pairwise key hierarchy takes a Pairwise Master Key and generates a Pairwise Transient Key. The PTK is partitioned into EAPOL-Key MIC and Encryption keys, and temporal keys used by the MAC to protect unicast communication between the Authenticator's and Supplicant's respective STAs. Pairwise keys are used between a single Supplicant and a single Authenticator.

Informative Note: In an ESS, the Pairwise Master Key results from authentication between the Supplicant and Authentication Server involved. This is often but not always a fresh key. An EAP authentication method normally has a Master Key generated by the authentication. In this case the PMK is derived from the Master Key. This key generation is normally carried out independently and simultaneously on the Authentication Server and the Supplicant, based on information that was communicated between the Authentication Server and the Supplicant during authentication. Each EAP method may derive the PMK from the Master Key in a different way.

If the protocol between the Authenticator or AP and Authentication Server is RADIUS then the MS-MPPE-Recv-Key attribute (vendor-id = 17; see RFC 2548 Section 2.4.3) is used to transport the Pairwise Master Key (PMK) to the AP. If the RADIUS Session-Timeout value is defined, the PMK and any derived keys shall not be used any longer than

$$\text{Session-Timeout} + (\text{reAuthMax} \times \text{dot1xAuthTxPeriod})$$

seconds. dot1xAuthTxPeriod is defined by IEEE 802.1X, while reAuthMax is an IEEE 802.11 MIB variable defined in Annex D. When RADIUS is used, and when the Radius Session-Timeout attribute is not in the RADIUS Accept message, the PMK lifetime is infinite.

Informative Note: If the authenticated key management protocol is RSN-PSK then a 256-bit pre-shared key is configured into the STA and AP. The method used to configure the PSK is outside this specification, but one method is via user interaction. The pre-shared key is used directly as the PMK.

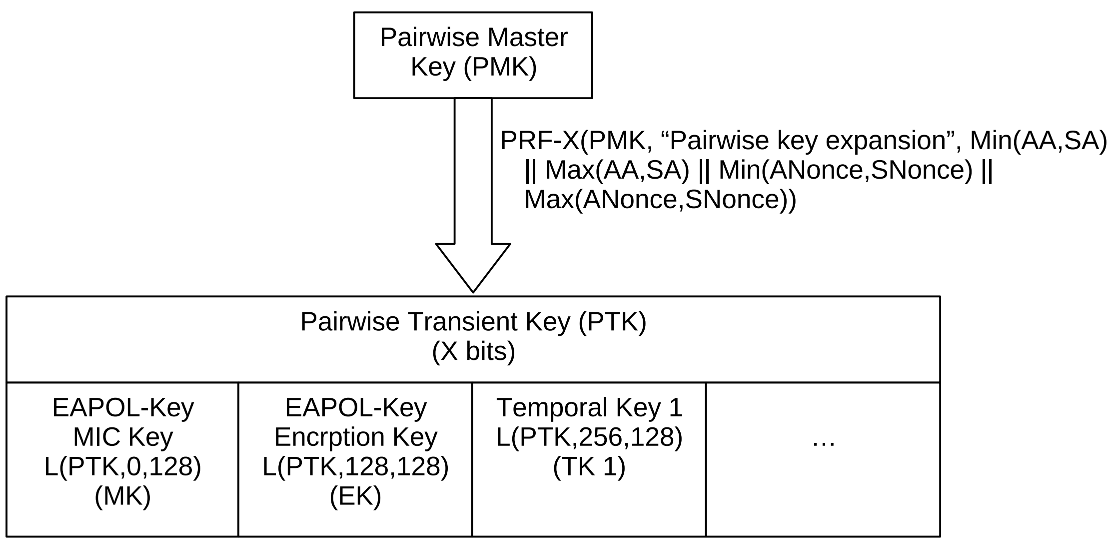


Figure 46—Pairwise key hierarchy

Here

- SNonce shall be a random or pseudo-random value contributed by the IEEE 802.1X Supplicant;.
- ANonce shall be a random or pseudo-random value contributed by the IEEE 802.1X Authenticator.
- The *Pairwise Transient Key* (*PTK*) shall be derived from the PMK by

PTK ← PRF-X(PMK, “Pairwise key expansion”, Min(AA,SA) || Max(AA, SA) || Min(ANonce,SNonce) || Max(ANonce,SNonce))

TKIP uses X = 512, while CCMP, WRAP, and WEP use X = 384. The Min and Max operations are with respect to lexicographic ordering of IEEE 802 addresses and the bit strings comprising the nonces, represented as in 7.1.1.

Informative Note: ANonce is taken from the Key Counter on the Authenticator whenever a new Pairwise TK is derived. ANonce is used so the inputs to PRF are different for each PMK. If a station re-associates to the same AP, a different ANonce value is used for the derivation of a new TK set.

Informative Note: SNonce is a nonce taken from the Key Counter on the Supplicant; its value is taken when a PTK is instantiated and is sent to the PTK Authenticator.

Informative Note: The Authenticator and Supplicant normally derive a PTK only once per association. A Supplicant or an Authenticator may use the 4-way handshake to derive a new PTK. This is required only after a TKIP data integrity failure. Both the Authenticator and Supplicant create a new nonce value for each 4-way handshake instance.

- The EAPOL-Key MIC key (MK) shall be computed as the first 128 bits (bits 0-127) of the PTK:

$$\text{MK} \leftarrow \text{L(PTK, 0, 128)}$$

The MK is used by IEEE 802.1X to provided data origin authenticity in the 4-way handshake and Group key distribution messages.

- The EAPOL-Key Encr. Key (EK) shall be computed as bits 128-255 of the PTK:

$$\text{EK} \leftarrow \text{L(PTK, 128, 128)}$$

The EK is used by IEEE 802.1X to provide confidentiality in the 4-way handshake and Group key distribution messages.

- Temporal Key 1 (TK1) shall be computed as bits 256-383 of the PTK:

$$\text{TK1} \leftarrow \text{L(PTK, 256, 128)}$$

TK1 shall be configured by IEEE 802.1X into IEEE 802.11 via the MLME-SETKEYS.request, to be consumed in the pairwise key cipher suite; interpretation of this value is cipher suite specific.

- Temporal Key 2 (TK2), if derived, shall be computed as bits 384-511 of the PTK:

$$\text{TK2} \leftarrow \text{L(PTK, 384, 128)}$$

TK2 shall be configured by IEEE 802.1X into IEEE 802.11 via the MLME-SETKEYS.request, to be consumed in the pairwise key cipher suite; interpretation of this value is cipher suite specific.

8.5.1.3 Group key hierarchy

The Group key hierarchy uses PRF-128 or PRF-256 to derive a group key. Figure 47 depicts the relationship among the keys of the Group key hierarchy. The Group key hierarchy takes a Group Master Key and generates a Group Transient key. The GTK is partitioned into temporal keys used by the MAC to protect broadcast/multicast communication. Group Keys are used between a single Authenticator and all Supplicants authenticated to that Authenticator. The Authenticator may derive new Group Transient Keys when it wants to update the Group temporal keys.

The Group Master Key (GMK) shall be 256 bits. It is used to derive the Group key hierarchy. The GMK shall be initialized using a cryptographically secure random number.

Any GMK must be re-initialized at a time interval configured into the AP, to reduce the exposure of data if the GMK is ever compromised.

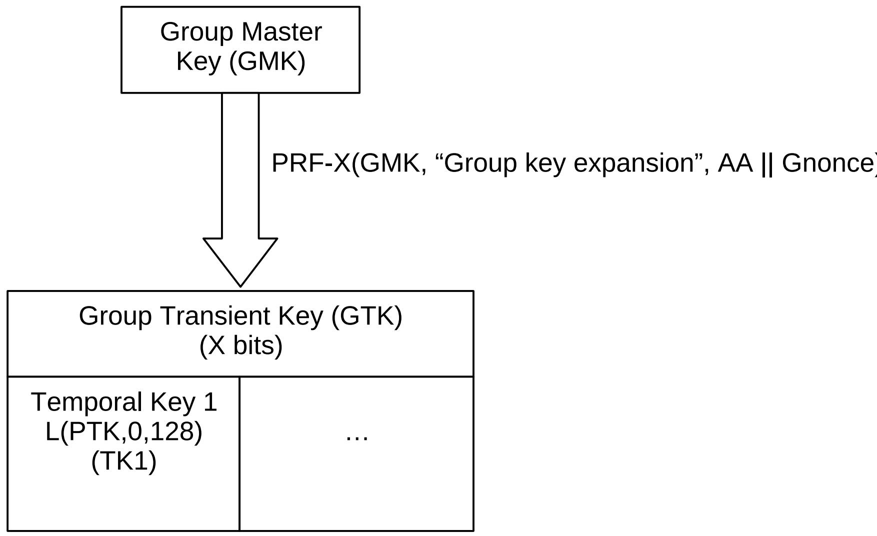


Figure 47—Group key hierarchy

Here

- GNonce shall be a random or pseudo-random value contributed by the IEEE 802.1X Authenticator.

- The *Group Transient Key* (*GTK*) shall be derived from the GMK by

  $$GTK \leftarrow \text{PRF-X(GMK, “Group key expansion” } \| \text{ AA } \| \text{ GNonce)}$$

  TKIP uses X = 256, while CCMP, WRAP, and WEP use X = 128. AA is represented as an IEEE 802 address and GNonce as a bit string as defined in 7.1.1.

- Temporal Key 1 (TK1) shall be bits 0-127 of the GTK:

  $$TK1 \leftarrow L(GTK, 0, 128)$$

  IEEE 802.1X configures TK1 into IEEE 802.11 via the MLME-SETKEYS.request, and IEEE 802.11 uses this key. Its interpretation is cipher suite specific.

- Temporal Key 2 (TK2), if derived, shall be bits 128-255 of the GTK:

  $$TK2 \leftarrow L(GTK, 128, 128)$$

  IEEE 802.1X configures TK1 into IEEE 802.11 via the MLME-SETKEYS.request, and IEEE 802.11 uses this key. Its interpretation is cipher suite specific.

  Informative Note: The Authenticator may update the Group key for a number of reasons:

1. The Authenticator may change the GTK on disassociation or Deauthentication of a STA..
2. A TKIP integrity failure shall trigger a Group key update.
3. A management event can trigger a Group key update.

8.5.2 EAPOL-KEY messages

IEEE 802.11 uses EAPOL-Key messages to exchange information between STAs' Supplicants and Authenticators that result in cryptographic keys and synchronization of security association state. EAPOL-Key messages are used to implement two different exchanges:

- 4-way handshake, to confirm that the PMK between associated STAs are the same and is live.
- The group key handshake, to update the GTK at the STA.

When used by an RSN, the RSN key descriptor carried by EAPOL-Key messages differs from IEEE 802.1X Clause 7.6, because it needs to convey different information and replaces the IEEE 802.1X Key descriptor.

The bit and octet convention for fields in the EAPOL-Key message are defined in IEEE 802.1X Clause 7.1.

Descriptor Type – 1 octet	
Key Information – 2 octets	Key Length – 2 octets
Replay Counter – 8 octets	
Key Nonce – 32 octets	
EAPOL-Key IV – 16 octets	
Key RSC – 8 octets	
Key ID – 8 octets	
Key MIC – 16 octets	
Key Material Length – 2 octets	Key Data – n octets

Figure 48—EAPOL-Key descriptor

Descriptor Type. This field is one octet and has a value of 254, identifying RSN Key Descriptor.

Key Information. This field is two octets and specifies characteristics of the key.

3 bits Key Descriptor Version	1 bit Key Type	2 bits Key Index	1 bit Install	1 bit Key Ack	1 bit Key MIC	1 bit Secure	1 bit Error	1 bit Request	4 bits Reserved

Figure 49—Key information bit layout

The bit convention used is as in 7.1 of IEEE 802.1X.

- Key Description Version Number (bits 0-2): specifies the Key descriptor version type.
  1. Type 1 indicates
     a) HMAC-MD5 is the EAPOL-Key MIC;
     b) RC4 is the EAPOL-Key encryption algorithm used to protect the distributed GTK.
  2. Type 2 indicates.
     a) AES-CBC-MAC is the EAPOL-Key MIC;
     b) HMAC-SHA1 is the EAPOL-Key encryption algorithm used to protect the distributed GTK. HMAC is defined in RFC 2104, and SHA1 by FIPS-180-1. The output of the HMAC-SHA1 shall be truncated to 128-bits.
- Key Type (bit 4): specifies whether this EAPOL-Key message represents a Pairwise or a Group key.
  1. The value 1 indicates a Pairwise key
  2. The value 0 indicates a Group key.
- Key Index (bits 5 and 6): specifies the key id of the temporal key of the key derived from the message. The value of this shall be zero (0) if the value of Key Type (bit 4) is Pairwise (1). The Key Type and Key Index shall not both be 0 in the same message.

  Group keys shall not use key id 0. This means that key ids 1 to 3 are available to be used to identify Group keys. This document recommends that implementations reserve key ids 1 and 2 for Group Keys, and that key id 3 is not used.

  The Key Type and Key Index shall not both be 0 in the same message.
- Bit 7 is the Install flag.
  1. If the value of Key Type (bit 4) is Pairwise (1), then
     a. The value 1 means the IEEE 802.1X component shall configure the temporal keys TK1 and TK2 derived from this message into its IEEE 802.11 STA.
     b. The value 0 means the IEEE 802.1X component shall not configure the temporal keys into the IEEE 802.11 STA.

2. If the value of Key Type (bit 4) is Group (0), then

   a. The value 1 means the IEEE 802.1X component shall configure the temporal keys TK1 and TK2 derived from this message into its IEEE 802.11 STA for both transmission and reception.

   b. The value 0 means IEEE 802.1X component shall configure the temporal keys TK1 and TK2 derived from this message into its IEEE 802.11 STA for reception only.

- Ack (bit 8): This bit is set in messages from the Authenticator if an EAPOL-Key message is required in response to this message, and clear otherwise. The Supplicant's response to this message shall use the same replay counter as this message.

- MIC (bit 9): this bit is set if a MIC is in this EAPOL-Key message, and it is clear if this message contains no MIC.

- Secure (bit 10): this bit is set once the initial key exchange is complete. That is, the secure bit in the EAPOL-Key message is used to inform when the pairwise key exchange is complete and the first Group Key Handshake is complete. It shall be initialized to 0 or not secure at the beginning of any 4-Way Handshake.

  The Authenticator shall set this bit to 1 in the final EAPOL-Key message that the Supplicant with the data needed to complete its initialization. At this point the Authenticator shall set the bit in all EAPOL-Key messages it sends until it no longer considers the link secure.

  The Supplicant will set the secure bit when it considers the link secure, which is when it has accepted enough keys to initialize the link. The number of keys should match the negotiated ciphers e.g. if a unicast and multicast cipher is negotiated then a Pairwise and Group key must be sent before the link is considered secure. The Supplicant shall clear the secure bit when it considers the link no-longer secure.

  The Supplicant and Authenticator shall consider the link insecure after a TKIP integrity error but prior to keys being re-established.

  Informative Note: The Supplicant and Authenticator initialize the secure bit to zero. Normally the Authenticator sets the secure bit when it sends the first Group key message to the Supplicant and the Supplicant sets the secure bit on receiving the first Group key message. The Supplicant clears the secure bit on receiving a TKIP integrity error from the MAC or on receiving an EAPOL-Key message with the secure bit cleared. The Authenticator clears the secure bit on receiving a TKIP integrity error from the Supplicant or from its STA.

- Error (bit 11): A Supplicant sets this bit to report that a MIC failure occurred in a TKIP MSDU. A Supplicant shall set this bit only when the Request (bit 12) is set.

- Request (bit 12): The Supplicant sets this bit to request that the Authenticator initiate either a 4-way or group key handshake. The Supplicant shall not set this bit in on-going 4-way handshakes, i.e., the Ack bit (bit 8) shall not be set in any message with the Request bit set. The Authenticator shall never set this bit.

  If the EAPOL-Key message with request bit set has a Key Type of Pairwise key, the authenticator shall initiate a 4-way handshake. If the EAPOL-Key message with request bit set has a key type of Group key, the authenticator shall change the Group key, initiate a 4-way handshake with the Supplicant and then execute the Group key handshake to all Supplicants.

  Informative Note: The Supplicant shall request a new key in response to any TKIP MIC failure.

- Reserved (bits 13-15). The sender shall set them to 0, and the receiver shall ignore the value of these bits.

Key Length. This field is two (2) octets in length, represented as an unsigned binary number. The value defines the length in octets of the key to configure into IEEE 802.11.

Informative Note: The rationale for this design is to hide from IEEE 802.1X the structure of the keys consumed by IEEE 802.11.

Informative Note: For Group Keys, the Key Data Length will be the same as the Key Length field for Key Descriptor Version 1 and Key Length + 8 octets for Key Descriptor Version 2.

Key Replay Counter. This field is eight (8) octets, represented an unsigned binary number, and is initialized to 0 when the PMK is established. The Supplicant shall use the replay counter in the received EAPOL-Key message when responding to an EAPOL-Key message. It carries a sequence number that the protocol uses to detect replayed EAPOL-Key messages.

The Supplicant and Authenticator shall track the Replay Counter per association. The replay counter shall be initialized to 0 on (re)association. The Authenticator shall increment the replay counter on each EAPOL-Key message.

When replying to a message from the Authenticator the Supplicant should use the replay counter received from the Authenticator. The Authenticator should use this to identify invalid messages to silently discard. The Supplicant should also use the replay counter and ignore EAPOL-Key messages with a replay counter smaller than any received in a valid message. The local replay counter should not be updated until the after EAPOL-Key MIC is checked and is valid. This means that the Supplicant never updates the replay counter for the first message in the 4-way handshake, as it includes no MIC. This implies the Supplicant must allow for re-transmission of the first message when checking for the replay counter of the third message.

The Supplicant shall maintain a separate replay counter for sending request EAPOL-Key messages to the Authenticator; the Authenticator also shall enforce monotonicity of a separate replay counter to filter received EAPOL-Key Request messages.

Informative Note: The Replay Counter does not play any role beyond a performance optimization in the 4-way handshake. In particular, replay protection is provided by selecting a never-before-used nonce value to incorporate into the PTK. It does, however, play a useful role in the Group key handshake.

Key Nonce. This field is thirty two (32) octets. It conveys the ANonce or GNonce from the Authenticator and the SNonce from the Supplicant. It may contain 0 if a Nonce is not required to be sent.

Key IV. This field is sixteen (16) octets. It contains the IV used with the key encrypting the Group Key. It may contain 0 when an IV is not required, i.e., when the message specifies a pairwise key. It should be initialized by taking the current value of the global Counter and then incrementing the counter. Note that only the lower sixteen octets of the counter value will be used.

Key RSC. This field is eight octets in length. It contains the receive sequence counter (RSC) for the key being installed in IEEE 802.11. It is only used in message 3 of the 4-way handshake and the first message of the Group key update, where it is used to synchronize the replay state. It shall contain 0 in other messages. If the key RSC is less than eight octets in length the remaining octets shall be set to 0. The least significant octet of the IV should be in the first octet of the Key RSC.

Informative Note: The Key RSC for TKIP is the TSC in the first 6 octets.

KeyRSC 0	KeyRSC 1	KeyRSC 2	KeyRSC 3	KeyRSC 4	KeyRSC 5	KeyRSC 6	KeyRSC 7
TSC0	TSC1	TSC2	TSC3	TSC4	TSC5	0	0

Informative Note: The Key RSC for WEP should be 0.

Key ID. This field is eight (8) octets in length. It is reserved and set to 0.

Key MIC. This field is sixteen octets (16) in length when the Key Descriptor Version field is 1 or 2. The EAPOL-Key MIC is a MIC of the EAPOL packet, from and including the EAPOL protocol version field, to and including the EAPOL-Key Material field with the EAPOL-Key MIC field set to 0 after any key material field is encrypted. If the Key data field contains a Group Key, the GTK is encrypted prior to calculation of the MIC.

Key Descriptor Version 1: HMAC-MD5; RFCs 2104 and 1321 together define this function, and Annex F.3 provides a reference implementation for it.

Key Descriptor Version 2: HMAC-MD5.

Key Data Length. This field is two (2) octets in length, taken to represent an unsigned binary number. This represents the length of the Key Data field in octets.

For Pairwise Keys, the Key Data Length value will be zero (0) in messages 1 and 4 of the 4-way handshake, and will be the length in octets of RSN IEs conveyed in the Key Data field in messages 2 and 3.

For Group Keys, the Key Data Length will be the same as the Key Length field.

Key Data. For EAPOL-Key messages specifying Pairwise Keys the Key Data field will contain the RSN information element in message 2 and 3 of the 4-way handshake and nothing for message 1 and 4.

For Pairwise keys this field contains the RSN information element contents (from and including the RSN element id) and the Key Data Length is set to the length of the information element contents for message 2 and 3 in the 4-way handshake. In message 1 and 4 this field is empty and the Key Data Length is 0. The RSN information element will not be encrypted when it is sent in the EAPOL-Key message.

The Supplicant should insert the RSN IE it sent in its (re)associate request into the second message of the 4-way handshake. On receipt of the second message the Authenticator shall bit-wise compare this against the RSN IE received in the IEEE 802.11 request.

The Authenticator should insert the RSN IE it sent in its Beacon or Probe Response into the third message of the 4-way handshake. On receiving the third message, the Supplicant shall bit-wise compare the RSN IE against the RSN IE received in the Beacon or Probe Response.

In either case, if the values do not match, then the receiver shall consider the RSN IE modified and shall use the MLME-DEAUTHENTICATE.request to break the association. A security error should be logged at this time.

For Group TKs this field contains the encrypted GTK.

Note that when checking the RSN information element the length of the RSN information element received in the beacon or probe response and sent in the associate request must be checked against the length of the RSN information element specified in EAPOL-Key Data Length.

Key Descriptor Version 1: RC4 is used to encrypt the Key Data field using the EK field from the derived PTK. No padding shall be used. The encryption key is generated by concatenating the EAPOL-Key IV field and the EK. The first 256 bytes of the RC4 key stream shall be discarded following RC4 stream cipher initialization with the EK, and encryption begins using the 257^{th} key stream byte.

Key Descriptor Version 2: AES Key Wrap, defined in RFC 3394, shall be used to encrypt the key material field using the EK field from the derived PTK. The key wrap default initial value shall be used.

8.5.2.1 EAPOL-Key message notation (Informative)

The following notation will often be used throughout to represent EAPOL-Key messages:

EAPOL-Key(S, M, A, T, N, K, KeyRSC, ANonce/SNonce, GNonce, MIC, GTK)

where the arguments are:

- S: Initial Key exchange is complete. This is the EAPOL-Key Information Secure bit.
- M: MIC is available in message. This should be set in all messages except the first 4-way handshake message. This is the EAPOL-Key Information Key MIC bit.
- A: Response is required to this message. Used when the receiver should respond to this message. This is the EAPOL-Key Information Key Ack bit.
- T: Tx/Rx for Group key and Install/Not install for Pairwise key. This is the EAPOL-Key Information Tx/Rx Flag bit.
- N: Key Index. Specifies which index should be used for this Group Key. Index 0 shall not be used for Group keys. This is the EAPOL-Key Information key index bits.
- K: Key type - P (Pairwise), G (Group). This is the EAPOL-Key Information Key Type bit.
- KeyRSC: Key RSC. This is the EAPOL-Key KeyRSC field.
- ANonce/SNonce/GNonce: Authenticator/Supplicant/Group Nonce. This is the EAPOL-Key Key Nonce field.
- MIC: Integrity check which is generated using the EAPOL-Key MIC Key. This is the EAPOL-Key MIC field.
- GTK: Group temporal key which is encrypted using the EAPOL-Key Encryption Key. This is the EAPOL-Key Data field.

8.5.3 4-way handshake

RSN defines an IEEE 802.1X protocol called the 4-way handshake. The 4-way handshake confirms the liveness of the STAs communicating directly with each other over the IEEE 802.1l link, guarantees the freshness of the their shared session key, binds the PMK to the MAC addresses of the communicating STAs, and synchronizes the usage of the key to secure the IEEE 802.11 link. The handshake completes the IEEE 802.1X authentication process. The information flow of the 4-way handshake is

1. Authenticator → Supplicant: EAPOL-Key(0,0,1,0,0,P,0,ANonce,0,0)

2. Supplicant → Authenticator: EAPOL-Key(0,1,0,0,0,P,0,SNonce,MIC,RSN IE)

3. Authenticator → Supplicant: EAPOL-Key(0,1,1,1,0,P,IV,ANonce,MIC,RSN IE)

4. Supplicant → Authenticator: EAPOL-Key(0,1,0,0,0,P,0,0,MIC,0)

Here

- EAPOL-Key(·) denotes an EAPOL-Key message conveying the specified argument list, using the notation introduced in 8.5.2.1.
- ANonce is a nonce the Authenticator contributes. ANonce has the same value in messages 1 and 3.
- SNonce is a nonce from the Supplicant. It assumes the same values in messages 2 and 4.
- P means the pairwise bit is set.
- MIC is computed over the body of the containing EAPOL-Key message (with the MIC field first zeroed before the computation) using the key MK defined in 8.5.1.2.
- RSN IE represents the appropriate RSN IEs.

Informative Note: While the MIC calculation is the same in each direction the Ack bit is different in each direction It is set in messages from the Authenticator and not set in messages from the Supplicant. 4-way handshake requests from the Supplicant have the Request bit set. The Authenticator and Supplicant must check these bits to stop reflection attacks.

8.5.3.1 Message 1

Message 1 uses of the following values for each of the EAPOL-Key message fields

Descriptor Type = 254

Key Information.
- Version = 1 (RC4 encryption with HMAC-MD5) or 2 (AES-128-CBC encryption with AES-128-CBC-MAC)
- Key Type = 1 (Pairwise)
- Key Index = 0 – Pairwise keys use KeyID 0
- Install flag = 0
- Key Ack = 1
- Key MIC = 0
- Secure = 0
- Error = 0
- Request = 0
- Reserved = 0 – unused by this protocol version

Key Length = 16 – all 802.11 keys are 16 octets in length

Key Replay Counter = *n* – to allow Authenticator to match the right Message 2 from Supplicant

Key Nonce = ANonce

Key IV = 0 – unused by the 4-way handshake

Key RSC = 0

Key ID = 0 – reserved

Key MIC = 0

Key Data Length = 0

Key Data = 0.

The Authenticator sends Message 1 to the Supplicant. On reception of message 1, the Supplicant determines whether the Replay Counter has been used before with the current security association. If the Replay Counter is less than or equal the current local value, the Supplicant discards the message. Otherwise the Supplicant

1. generates a new nonce SNonce,

2. derives PTK, and

3. constructs Message 2.

8.5.3.2 Message 2

Message 2 uses of the following values for each of the EAPOL-Key message fields

Descriptor Type = 254

Key Information.
- Version = 1 (RC4 encryption with HMAC-MD5) or 2 (AES-128-CBC encryption with AES-128-CBC-MAC) – same as Message 1
- Key Type = 1 (Pairwise) – same as Message 1
- Key Index = 0 – same as Message 1
- Install flag = 0
- Key Ack = 0
- Key MIC = 1
- Secure = 0 – Same as Message 1
- Error = 0 – Same as Message 1
- Request = 0 – Same as Message 1
- Reserved = 0 – unused by this protocol version

Key Length = 16 – same as Message 1

Key Replay Counter = *n* – To let the Authenticator knows which Message 1 this corresponds to.

Key Nonce = SNonce

Key IV = 0 – unused by the 4-way handshake

Key RSC = 0

Key ID = 0 – reserved

Key MIC = MIC(MK, EAPOL) – MIC computed over the body of this EAPOL-Key message with the Key MIC field first initialized to 0.

Key Data Length = length in octets of included RSN IE

Key Data = included RSN IE – in a BSS, the STA's RSN IE

The Supplicant sends Message 2 to the Authenticator.

On reception of message 2, the Authenticator checks that the Replay Counter corresponds to the outstanding Message 1. If not, it silently discards the message. Otherwise, the Authenticator

1. derives PTK and

2. verifies the Message 2 MIC. If the MIC is not valid, the Authenticator silently discards the packet. If the MIC is valid, the Authenticator

3. checks that the RSN IE bit-wise matches that from the (re)association request message. If these are not exactly the same, the Authenticator uses MLME-DEAUTHENTICATE.request to terminate the association. If they do match bit-wise, the Authenticator

4. constructs message 3.

8.5.3.3 Message 3

Message 3 uses of the following values for each of the EAPOL-Key message fields

Descriptor Type = 254

Key Information.
- Version = 1 (RC4 encryption with HMAC-MD5) or 2 (AES-128-CBC encryption with AES-128-CBC-MAC) – same as Message 1
- Key Type = 1 (Pairwise) – same as Message 1
- Key Index = 0 – Same as Message 1
- Install = 0/1 – 0 only if AP does not support key mapping keys
- Key Ack = 1
- Key MIC = 1
- Secure = 0 (Group key handshake to come) or 1 (no group key handshake)
- Error = 0 – same as Message 1
- Request = 0 – same as Message 1
- Reserved = 0 – unused by this protocol version

Key Length = 16

Key Replay Counter = *n* – which transaction does this belong to?

Key Nonce = ANonce – same as Message 1

Key IV = 0 – unused by the 4-way handshake

Key RSC = starting sequence number Authenticator's STA will use in packets protected by PTK (normally 0)

Key ID = 0 – reserved

Key MIC = MIC(MK, EAPOL) – MIC computed over the body of this EAPOL-Key message with the Key MIC field first initialized to 0.

Key Data Length = length in octets of included RSN IE

Key Data = included RSN IE – in a BSS, the AP's Beacon/Probe RSN IE

The Authenticator sends Message 3 to the Supplicant.

On reception of message 3, the Supplicant verifies the Replay Counter is not an already used value or the ANonce differs from that in Message 1. If so, it silently discards the message. Otherwise, the Supplicant

1. verifies the Message 3 MIC. If this is invalid, the Supplicant silently discards. Otherwise the Supplicant

2. updates the last-seen value of the Replay Counter,

3. constructs Message 4,

4. sends Message 4 to the Authenticator, and

5. uses the MLME-SETKEYS.request to configure the IEEE 802.11 to send and receive class 3 unicast MPDUs protected by the PTK,

Informative Note: after configuring the PTK into the IEEE 802.11 MAC, the STA must still be able to receive Message 3 in the clear, to handle the case where its Message 4 does not arrive at the AP.

Informative Note: If Message 4 is lost and the Authenticator retries Message 3, then the STA will resend the response protected by the temporal key as well as the MK.

8.5.3.4 Message 4

Message 4 uses of the following values for each of the EAPOL-Key message fields

Descriptor Type = 254

Key Information.
- Version = 1 (RC4 encryption with HMAC-MD5) or 2 (AES-128-CBC encryption with AES-128-CBC-MAC) – same as Message 1
- Key Type = 1 (Pairwise) – same as Message 1
- Key Index = 0
- Install = 0
- Key Ack = 0 – This is the last message
- Key MIC = 1
- Secure = 0 or 1 – same as Message 3
- Error = 0
- Request = 0
- Reserved = 0 – unused by this protocol version

Key Length = 16

Key Replay Counter = *n* – which transaction does this belong to?

Key Nonce = 0 – not used in Message 4.

Key IV = 0 – unused by the 4-way handshake

Key RSC = starting sequence number Supplicant's STA will use in packets protected by PTK (normally 0)

Key ID = 0 – reserved

Key MIC = MIC(MK, EAPOL) – MIC computed over the body of this EAPOL-Key message with the Key MIC field first initialized to 0.

Key Data Length = 0

Key Data = 0.

The Supplicant sends Message 4 to the authenticator. Note that it is protected by the agreed upon temporal key as well as the PTK.

On receipt, the Authenticator verifies that the Replay Counter value is one that it used on this 4-way handshake; if it is not, it silently discards the message. Otherwise, the Authenticator

1. checks the MIC, and if invalid, silently discards the packet; if it is valid, the Authenticator otherwise

2. uses the MLME-SETKEYS.request to configure the PTK into the IEEE 802.11 MAC.

3. The Authenticator finally updates the Replay Counter, so that it will use a fresh value if a rekey becomes necessary.

8.5.3.5 4-way handshake implementation considerations

If the Authenticator does not receive a reply to its messages, its AP shall retry up to three times at one second intervals; if it still has not received a response after these retries, then the Authenticator's AP should disassociate the STA.

If the STA does not receive the initial message when it expects to, it should disassociate, deauthenticate, and try another AP/STA.

Informative Note: The timeout should be larger than the short retry timeout.

The Authenticator should ignore EAPOL-Key messages it is not expecting in reply to messages it has sent or EAPOL-Key messages with the Ack bit set. This stops an attacker from sending the first message to the supplicant who responds to the Authenticator.

An implementation should save the EAPOL-Key MIC key MK and EAPOL-Key encryption key TK beyond the 4-way handshake, as they are needed by the Group Key handshake and to recover from TKIP MIC failures.

The Supplicant uses the MLME-SETKEYS.request to configure the temporal keys TK1, TK2, … from 8.5.1 into its STA after sending Message 4 to the Authenticator.

Informative Note: If the RSN IE check for the second or third message fails, IEEE 802.1X should log an error and deauthenticate the peer.

Informative Note: The Supplicant should check that if the RSN IE specifies a unicast cipher is used then the 4-way handshake did specify that the Pairwise key is configured to the encryption/integrity engine.

8.5.3.6 Example 4-way handshake (Informative)

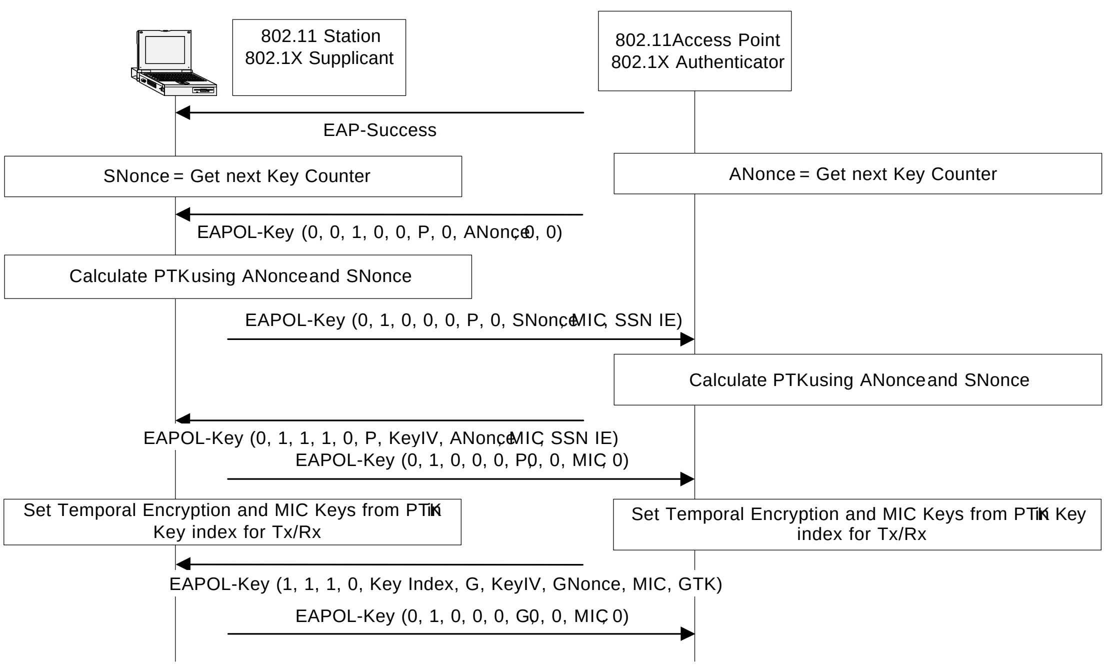


Figure 50—Example 4-way handshake

After IEEE 802.1X authentication per se completes by the AP sending an EAP-Success, the AP initiates two Key exchanges: the 4-way handshake and the Group key handshake. The 4-way handshake consists of:

1. The Authenticator sending an EAPOL-Key message containing an ANonce.
2. The Supplicant derives a PTK from ANonce and SNonce.
3. The Supplicant sends an EAPOL-Key message containing SNonce, the RSN information element from the (Re)associate request, and a MIC.
4. The Authenticator derives PTK from ANonce and SNonce and validates the MIC in the EAPOL-Key message.
5. The Authenticator sends an EAPOL-Key message containing ANonce, the RSN IE from its Beacon or Probe Response messages, MIC, and whether to install the temporal keys.
6. The Supplicant sends an EAPOL-Key message to confirm that the temporal keys are installed.

The AP typically follows this with the initial Group key update.

Informative Note: Step 6 could be eliminated from the protocol when Pairwise keys are not being used for encryption/integrity, but for consistency it has been included in all cases.

Informative Note: The "Initial exchange complete" bit is set in the last message from the Authenticator to the Supplicant to inform the Supplicant that the last key required to initialize the Supplicant has been sent. Once set the "Initial exchange complete" bit should be set in any EAPOL-Key messages from the Authenticator until a 4-way handshake is initiated.

8.5.3.7 4-way handshake analysis (Informative)

First we want to make the trust assumptions explicit. The protocol assumes the PMK is known only by the Supplicant's STA and the Authenticator's STA, and that the Supplicant's STA uses IEEE 802 address SA, and the Authenticator's STA uses IEEE 802 address AA. In many instantiations the RSN architecture immediately breaks the first assumption, since the IEEE 802.1X AS also knows the PMK. Therefore, we require additional assumptions (a) the AS does not expose the PMK to other parties, (b) the AS does not masquerade as the Supplicant to the Authenticator, (c) the AS does not masquerade as the Authenticator to the Supplicant, (d) the AS does not masquerade as the Supplicant's STA, and (e) the AS does not masquerade as the Authenticator's STA. The protocol also assumes this particular Supplicant/Authenticator pair are authorized to know this PMK and to use it in the 4-way handshake. If any of these assumptions are broken, then the protocol fails to provide any security guarantees.

The protocol also assumes that the AS delivers the correct PMK to the AP with IEEE 802 address AA, and that the non-AP STA with IEEE 802 address AP hosts the Supplicant that negotiated the PMK with the AS. None of the protocols defined by IEEE 802.11 and IEEE 802.1X permit the AS, the Authenticator, the Supplicant, or either STA to verify these assumptions.

The protocol supplies no mechanism to identify the correct PMK to use. This implies that a STA must negotiate a new PMK each time it visits an AP.

The PTK derivation step

PTK ← PRF-X(PMK, "Pairwise key expansion" || Min(AA,SA) || Max(AA, SA) ||
Min(ANonce,SNonce) || Max(ANonce,SNonce))

performs a number of functions:

- Including the AA and SA in the computation (1) binds the PTK to the communicating STAs and (2) prevents undetected man-in-the-middle attacks against 4-way handshake messages between the STAs with these two IEEE 802 addresses.

- If ANonce is randomly selected, including ANonce (1) guarantees the STA at IEEE 802 address AA that PTK is fresh, (2) that Messages 2 and 4 are live, and (3) uniquely identifies PTK as <AA, ANonce>.

- If SNonce is randomly selected, including SNonce (1) guarantees the STA at IEEE 802 address SA that PTK is fresh, (2) that Message 3 is live, and (3) uniquely identifies PTK as <SA, SNonce>.

Choosing the nonces randomly helps prevent pre-computation attacks. With unpredictable nonces, a man-in-the-middle attack that uses the Supplicant to pre-compute messages to attack the Authenticator cannot progress beyond Message 2, and a similar attack against the Supplicant cannot progress beyond Message 3. The protocol can be executed further if predictable nonces are used.

Message 1 delivers ANonce to the Supplicant and initiates negotiation for a new PTK. It identifies AA as the peer STA to the Supplicant's STA. If an adversary modifies either of the addresses or ANonce, the Authenticator will detect the result when validating the MIC in Message 2. Message 1 does not carry a MIC, as it is impossible for the Supplicant to distinguish this message from a replay without maintaining state of all security associations through all time (PMK might be a static key).

Message 2 delivers SNonce to the Authenticator, so it can derive the PTK. If the Authenticator selected ANonce randomly, Message 2 also demonstrates to the Authenticator that the Supplicant is live, the PTK is fresh, and that there is no man-in-the-middle, as the PTK includes the IEEE 802 MAC addresses of both. Inclusion of ANonce in the PKT derivation also protects against replay. The MIC prevents undetected modification of Message 2 contents.

Message 3 confirms to the Supplicant that there is no man-in-the-middle. If the Supplicant selected SNonce randomly, it also demonstrates that the PTK is fresh and that the Authenticator is live. The MIC again prevents undetected modification of Message 2.

Message 4 serves no cryptographic purpose.

Then the 4-way handshake uses a correct but unusual mechanism to guard against replay. As noted above, ANonce provides replay protection to the Authenticator, and SNonce to the Supplicant. In most session initiation protocols, replay protection is accomplished explicitly by selecting a nonce randomly and requiring the peer to reflect the received nonce in a response message. The 4-way handshake instead mixes ANonce and SNonce into the PTK, and replays are detected implicitly by MIC failures. In particular, the Replay Counter field appears to serve no cryptographic purpose in the 4-way handshake. Its presence is not detrimental, however, and it seems to play a useful role as a minor performance optimization for processing stale instances of Message 2. This replay mechanism is correct, but its implicit nature makes the protocol harder to understand than an explicit approach.

It is critical to the correctness of the 4-way handshake that at least one bit differs in each message. Within the 4-way handshake, Message 1 can be recognized as the only one with the MIC bit clear, meaning Message 1 does not include the MIC, while Messages 2-4 do. Message 3 differs from Message 2 by not asserting the Ack bit and from Message 4 by asserting the Ack Bit. Message 2 differs from Message 4 by including the RSN IE.

Request messages cannot be confused with 4-way handshake messages, since the former asserts the Request bit and 4-way handshake messages do not. Group key handshake messages cannot be mistaken for 4-way handshake messages, since they assert a different Key Type.

8.5.4 Group key handshake

The Authenticator uses the Group Key handshake to send a new Group Transient Key (GTK) to the Supplicant. The Authenticator may initiate this as the final stage of authenticating a Supplicant.

If the Authenticator is the GTK authenticator, and if the group key cipher suite is TKIP, the authenticator shall initiate the exchange if its AP detects a TKIP data integrity failure using the GTK, when a Supplicant disassociates or deauthenticates, or on a management event.

Authenticator → Supplicant: EAPOL(1,1,1,0,Key Id,G, RSC, GNonce, MIC,GTK)

Supplicant → Authenticator: EAPOL(1,0,0,0,G,0,0,MIC,0)

Here

- KeyId identifies the WEP key id the Authenticator's STA will use when sending traffic protected by the GTK.
- RSC denotes the last packet sequence number sent using the GTK.
- GTK denotes the GTK encrypted using the key EK defined in 8.5.1.
- MIC is computed over the body of the containing EAPOL-Key message (with the MIC field zeroed for the computation) using the key MK defined in 8.5.1.

Informative Note: The Supplicant may trigger a Group Key Update by sending an EAPOL-Key message with the Request bit set to 1 and by the type of the Group key bit.

An Authenticator shall do a 4-way handshake before a Group Key Update if both are required to be done.

Informative Note: The Supplicant does not require the GNonce but the Authenticator should send the Nonce it used to derive the GTK to help with interoperable issues. Rather, GNonce is useful for debugging.

Informative Note: The Authenticator cannot initiate the Group Key handshake until the 4-way handshake completes successfully.

If an AP cannot send the EAPOL-Key message containing a Group Key to a STA, the AP may queue the message. If the AP deletes the message, the AP should send a Deauthentication message and then delete the association state by setting the L2Failure event in the Authenticator state machine.

8.5.4.1 Message 1

Message 1 uses of the following values for each of the EAPOL-Key message fields

Descriptor Type = 254

Key Information.
Version Number = 1 (RC4 encryption with HMAC-MD5) or 2 (AES-128-CBC encryption with AES-128-CBC-MAC)
Key Type = 0 (Group)
KeyID = 1, 2, or 3
Install flag = 1
Key Ack = 1
Key MIC = 1
Secure = 1
Error = 0
Request = 0
Reserved = 0

Key Length = 16

Key Replay Counter = *n*

Key Nonce = GNonce

Key IV = version specific

Key RSC = last transmit sequence number for the GTK.

Key ID = 0 – reserved

Key MIC = MIC(MK, EAPOL)

Key Material Length = 32

Key Material = version specific

The Authenticator sends Message 1 to the supplicant.

Informative Note: To prevent replay attacks of packets sent prior to joining the BSS, the KeyRSC is sent with the GTK so that newly associated STAs start with the current value of the Group Key sequence counter. It may take a short time for the STA to get the current RSC from the AP, so packets affecting the value of the RSC may be sent between the current value and that obtained from the AP. Therefore some small window of vulnerability to replay attack necessarily exists.

On reception of Message 1, the Supplicant

1. verifies that the Replay counter has not yet been seen before, i.e., its value is strictly larger than that in any other EAPOL-Key message received thus far during this session.

2. verifies that the MIC is valid, i.e., it uses the MK that is part of the PTK to verify that there is no data integrity error.

3. uses the MLME-SETKEYS.request to configure the temporal GTK into its IEEE 802.11 MAC, and responds by creating and sending Message 2 of the Group Key handshake to the Authenticator and increment the Replay Counter.

Informative Note: The Authenticator must increment and use a new Replay Counter value on every Message 1 instance, even retries, because the Message 2 responding to an earlier Message 1 may have been lost. If the Authenticator did not increment the Replay Counter, the Supplicant will discard the retry, and no responding Message 2 will ever arrive.

8.5.4.2 Message 2

Message 2 uses of the following values for each of the EAPOL-Key message fields

Descriptor Type = 254

Key Information.

Version number = 1 (RC4 encryption with HMAC-MD5) or 2 (AES-128-CBC encryption with AES-128-CBC-MAC) – same as Message 1
Key Type = 0 (Group) – same as Message 1
KeyID = 1, 2, or 3 – same as Message 1
Install = 0
Key Ack = 0
Key MIC = 1
Secure = 1
Error = 0
Request = 0
Reserved = 0

Key Length = 16

Key Replay Counter = *n – same as Message 1*

Key Nonce = 0

Key IV = 0

Key MIC = MIC(MK, EAPOL)

Key Material Length = 0

Key Material = 0.

On reception of Message 2, the Authenticator

1. verifies that the Replay Counter matches one it has used in the Group Key handshake.
2. verifies that the MIC is valid, i.e., it uses the MK that is part of the PTK to verify that there is no data integrity error.

8.5.4.3 Group key distribution implementation considerations

If the authenticator does not receive a reply to its messages, its AP should retry up to three times at one second intervals; if it still has not received a response after this, then the Authenticator's AP should disassociate/deauthenticate the STA.

8.5.4.4 Example Group key distribution (Informative)

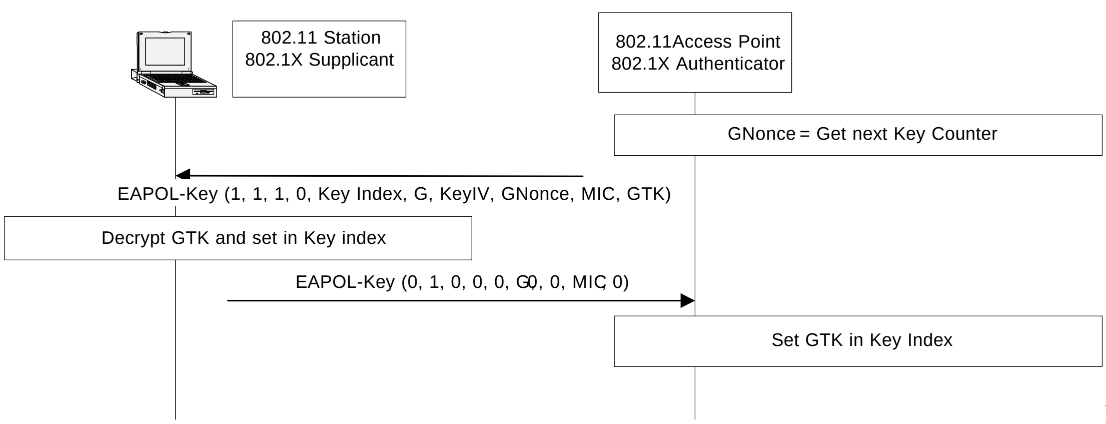


Figure 51—Example group key distribution

The Group key handshake state machine changes the Group key in use by the network. The following steps occur:

1. The Authenticator generates a new GTK. It encrypts the GTK and sends an EAPOL-Key message containing the GTK (Message 1), along with the last sequence number used with the GTK (RSC).
2. On receiving the EAPOL-Key message, the Supplicant validates the MIC, decrypts the GTK, and uses the MLME-SETKEYS.request primitive to configure the GTK and the RSC in its STA.
3. The Supplicant then constructs and sends an EAPOL-Key message in acknowledgement to the Authenticator.
4. On receiving the EAPOL-Key message, the Authenticator validates the MIC. If the GTK is not already configured into IEEE 802.11, after it has delivered the GTK to all associated STAs, it uses the MLME-SETKEYS.request primitive to configure the GTK into 802.11.

8.5.5 Supplicant key management state machine

There is one state machine for Supplicants. The Supplicant shall reinitialize the Supplicant state machine whenever its system initializes. A Supplicant enters the AUTHENICATION state on an event from the MAC that requests another STA to be authenticated. A Supplicant enters the STAKEYSTART state on receiving an EAPOL-Key messages from the Authenticator. If the MIC on any of the EAPOL-Key messages fails, the Supplicant silently discards the packet.

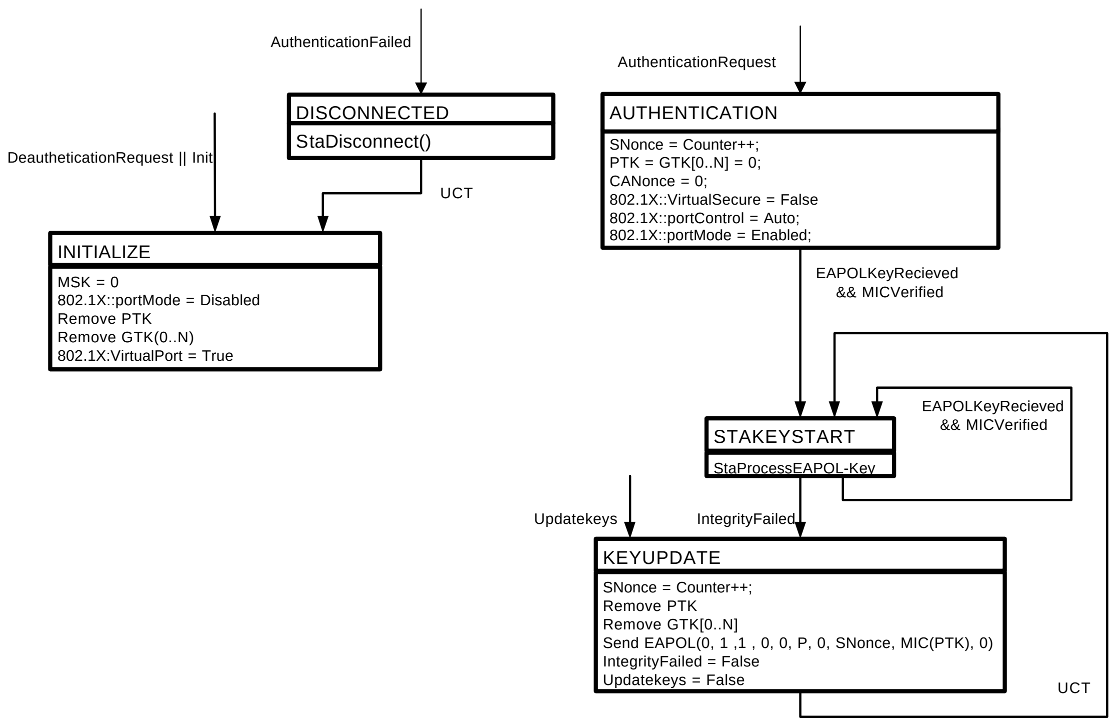


Figure 52—Supplicant key management state machine

UCT means the event triggers an immediate transition.

This state machine does not use timeouts, etc. The IEEE 802.1X state machine has timeouts that recover from Authentication failures, etc.

The Management entity will send an AuthenticationRequest event when it wants an Authenticator authenticated, this can be before or after the station associates to the AP. In an IBSS environment the event will be generated when a Probe Response is received.

8.5.5.1 Supplicant state machine states

DISCONNECTED: A STA's supplicant enters this state when IEEE 802.1X authentication fails. The supplicant executes StaDisconnect and enters the INITIALIZE state.

INITIALIZE: A STA's supplicant enters this state from the DISCONNECTED state, when it receives disassociate or Deauthentication messages, or when the STA initializes, causing the STA's supplicant to initialize the key state variables.

AUTHENTICATION: A STA's supplicant enters this state when it sends an IEEE 802.1X AuthenticationRequest to authenticate an SSID.

STAKEYSTART: A STA's supplicant enters this state when it receives an EAPOL-Key message. All the information to process the EAPOL-Key message is in the message and is described in procedure StaProcessEAPOL-Key.

KEYUPDATE: A STA's supplicant enters this state when its STA requires a key update from the authenticator. This may be because of a management event or because of a data integrity failure occurs. From this state the supplicant sends an EAPOL-Key message to the authenticator to update the transient keys. The Request bit shall be set.

8.5.5.2 Supplicant state machine variables

DeauthenticationRequest – The Supplicant set this variable to TRUE if the Supplicant's STA reports it has received disassociate or Deauthentication messages.

AuthenticationRequest – The Supplicant sets this variable to TRUE if its STA's IEEE 802.11 Management Entity reports it wants an SSID authenticated. This can be on association or at other times.

AuthenticationFailed – The Supplicant sets this variable to TRUE if the IEEE 802.1X authentication failed. The Supplicant uses the MLME-DISASSOCIATE.request to cause its STA to disassociate from the authenticator's STA.

EAPOLKeyReceived – The Supplicant sets this variable to TRUE when it receives an EAPOL-Key message.

IntegrityFailed **–** The Supplicant sets this variable to TRUE when its STA reports that a fatal data integrity error (e.g. Michael failure) has occurred.

> Informative Note: A Michael failure is not the same as MICVerified since IntegrityFailed is generated if the MAC integrity check fails, MICVerified is generated from validating the EAPOL-Key MIC. Note also the STA does not generate this event for CCMP or WRAP, since countermeasures are not required.

MICVerified **–** The Supplicant sets this variable to TRUE if the MIC on the received EAPOL-Key message verifies as correct. The Supplicant silently discards any EAPOL-Key message received with an invalid MIC.

Counter **–** The Supplicant uses this variable as a global counter used for generating nonces.

SNonce **–** This variable represents the Supplicant's nonce.

PTK **–** This variable represents the current PTK.

TPTK **–** This variable represents the current PTK until the third message of the 4-way handshake arrives and is verified.

GTK[] **–** This variable represents the current GTKs for each group key index.

PMK **–** This variable represents the current PMK.

802.1X::XXX **–** denotes another IEEE 802.1X state variables XXX not specified herein.

8.5.5.3 Procedures

STADisconnect. The Supplicant invokes this procedure to disassociate and deauthenticate its STA from the AP.

RemoveGTK – The Supplicant invokes this procedure to remove the GTK from its STA.

MIC(*x*) – The Supplicant invokes this procedure to compute a Message Integrity Code of the data *x*.

CheckMIC() **–** The supplicant invokes this procedure to verify a MIC computed by the MIC() function.

StaProcessEAPOL-Key – The Supplicant invokes this procedure to process a received EAPOL-Key message. The pseudo code for this procedure is:

```
StaProcessEAPOL-Key (S, M, A, T, N, K, RSC, ANonce, GNonce, MIC, GTK)
    TPTK ← PTK
    TSNonce ← 0
    UpdatePTK ← 0
    State ← UNKNOW
    if M = 1 then
        if Check MIC(PTK, EAPOL-Key message) fails then
            State ← FAILED
        else
            State ← MICOK
        endif
    endif
    if K = P then
        if State ≠ FAILED then
            if PSK exists then – PSK is a pre-shared key
                PMK ← PSK
            else
                PMK ← Master Session Key from 1X
            endif
            TSNonce ← SNonce
            TPTK ← Calc PTK(ANonce, TSNonce)
        endif
        if State = MICOK then
            PTK ← TPTK
            UpdatePTK ← TRUE
        endif
    else if State = MICOK then  -- K = G
        if GTK[N] ← Decrypt GTK succeeds then
            if Set GTK(N, T, RSC, GTK[N]) fails then
                invoke MLME-DEAUTHENTICATE.request
            endif
        else
            State ← FAILED
        endif
    else
        State ← FAILED
    endif
    if A = 1 and State ≠ FAILED then
        Send EAPOL(0, 1, 0, 0, 0, K, 0, TSNonce, 0, MIC(TPTK), 0)
    endif
    if UpdatePTK = 1 then
        if Set PTK(N, TRUE, RSC, PTK) fails then
            invoke MLME-DEAUTHENTICATE.request
    endif
    if State = MICOK and S = 1 then
        802.1X::VirtualSecure = TRUE
    endif
```

Here UNKNOWN, MICOK and FAILED are values of the variable State used in the Supplicant pseudo code. State is used to decide to do the key processing. MICOK is set when the MIC of the EAPOL-Key has been checked and is valid. FAILED is used when a failure has occurred in processing the EAPOL-Key message. UNKNOWN is the initial value of the State variable.

> Informative Note: A Supplicant shall only use Key Descriptor of type 254 and version 1 or 2 to and from RSN Access Points; it shall ignore other Key Descriptor types and Versions.

> Informative Note: EAPOL-Key messages with Key Type of Pairwise and a non-zero key index should be ignored.

> Informative Note: EAPOL-Key messages with Key Type of Group and an invalid key index should be ignored.

> Informative Note: The Replay Counter used by the Supplicant for EAPOL-Key messages that are sent in response to a received EAPOL-Key message must be the received Replay Counter.

> Informative Note: TPTK is used to stop attackers changing the PTK on the supplicant by sending the first message of the 4-way handshake. An attacker can still affect the 4-way handshake while the 4-way handshake is being carried out.

> Informative Note: The PMK will be supplied by the authentication method used with IEEE 802.1X if Pre-shared mode is not used.

> Informative Note: Invalid EAPOL-Key messages such as invalid MIC, Group Key without a MIC, etc. are ignored.

> Informative Note: A PTK is configured into the encryption/integrity engine depending on the Tx/Rx bit but if configured is always a transmit key. A GTK is configured into the encryption/integrity engine independent of the state of the Tx/Rx bit but whether the GTK is used as a transmit key is dependent on the state of the Tx/Rx bit.

CalcGTK(x) – Calculates the Group Transient Key (GTK) using GNonce as the nonce input to the PRF.

DecryptGTK(x) – Decrypt the GTK from the EAPOL-Key message

SetPTK/GTK(x) **–** Sets the PTK/GTK into the encryption/integrity engine

> Informative Note: On receiving the IEEE 802.1X EAP-Success message a Supplicant should compare the received keys and the ciphers specified in the RSN information element for consistency problems. E.g. the RSN information element specifies a unicast cipher but no Pairwise Key was configured into the encryption/integrity engine.

8.5.6 Authenticator key management state machine

There is one state diagram for the GTK Authenticator. In an ESS the GTK Authenticator will always be the AP, and in an IBSS environment will be a designated machine.

The state diagram in Figure 53 consists of three state machines:

1. The first state machine (PTK state machine) uses the DEAUTHENTICATE, DISCONNECTED, INITIALIZE, AUTHENTICATION, INITPMK, INITPSK, PTKSTART, PTKINITNEGOTIATING, UPDATEKEYS, MICFAILURE and UPDATEKEYS states. An instance of this state machine exists for each association and handles the initialization, 4-way handshake, tear-down, and general clean-up.

2. The second state machine (PTK Group Key state machine) uses the REKEYNEGOTIATING, KEYERROR and REKEYESTABLISHED states. An instance of this state machine exists for each association and handles transfer of the GTK to the associated client.

3. The third state machine (Group Key state machine) uses the SETKEYS and SETKEYSDONE states. A single instance of this state machine exists on the Authenticator. It changes the Group key when required, triggers all the PTK Group Key state machines and updates the IEEE 802.11 MAC in the Authenticator's AP when all STAs have the updated Group key.

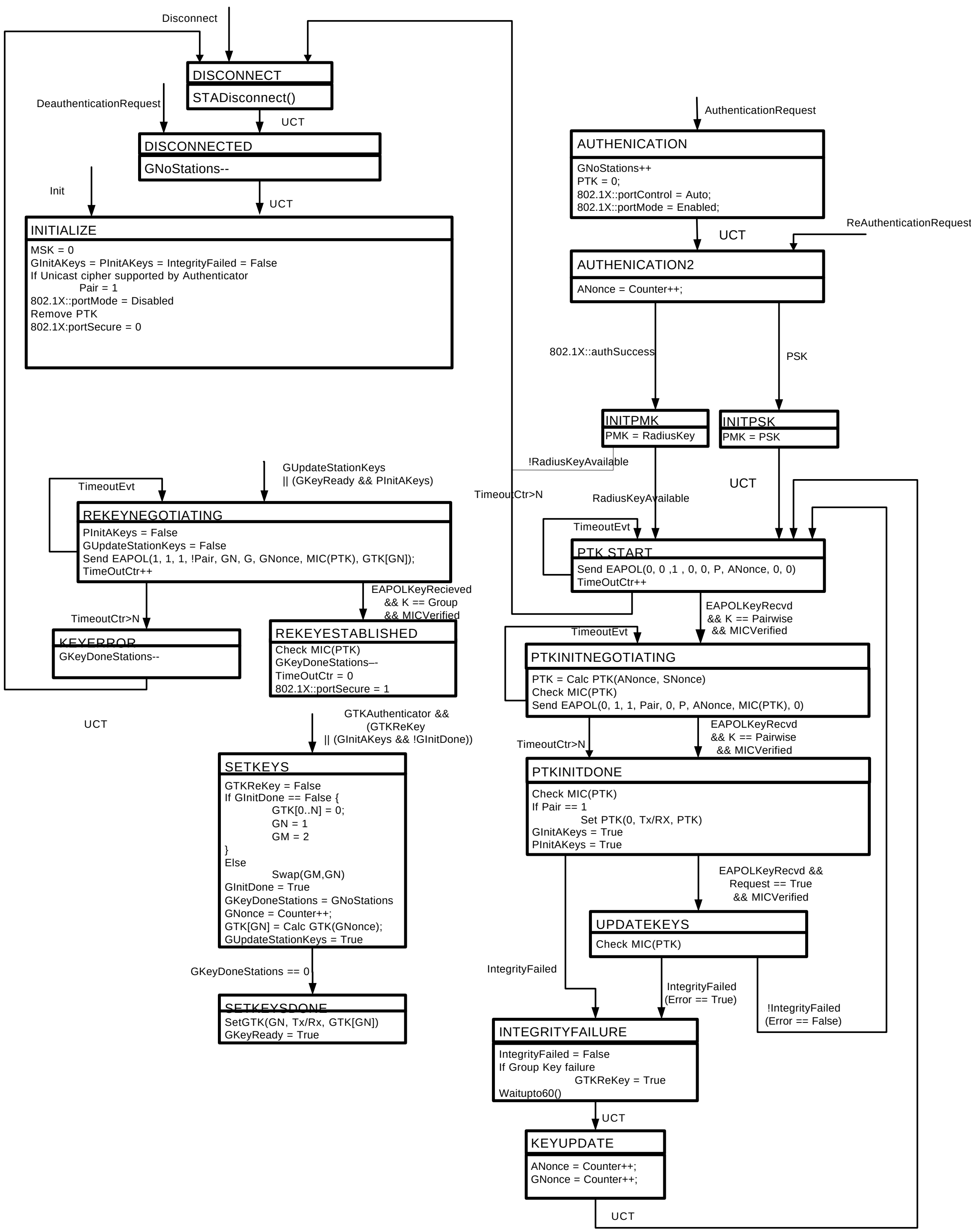


Figure 53—Authenticator state machine

Since there are two GTKs, responsibility for updating these keys is given to the Group Key state machine. That is, this state machine determines which GTK is in use at any time. When a first STA associates, the Group Key state machine has not been started and is started by *GInitAKeys* variable when the 4-way handshake completes. The Group Key state machine initializes the value of the Group Key and then triggers the PTK Group Key state machine, which actually sends the Group Key to the associated station.

When a second STA associates, the Group Key state machine is already initialized, and a Group Key is already available and in use. The PTK Group Key state machine is immediately triggered from the PTKINITDONE state and sends the current Group Key to the new station.

When the GTK is to be updated the *GTKReKey* variable is set. The SETKEYS state updates the Group Key and triggers all the PTK Group Key state machines that current exist—one per associated STA). Each PTK Group Key state machine sends the Group Key to its station. When all the stations have received the Group Key (or failed to receive the key), the SETKEYSDONE state is executed which updates the APs encryption/integrity engine with the new key.

Both the PTK state machine and the PTK Group Key state machine both use received EAPOL-Key messages as an event to change states. The PTK state machine only uses EAPOL-Key messages with the key type set to Pairwise key and the PTK Group Key state machine only uses EAPOL-Key messages with the key type set to Group key.

8.5.6.1 Authenticator state machine states

DEAUTHENTICATE: This state is entered is an EAPOL-Key message is received and fails its MIC check. It sends a Deauthentication message to the Access Point and enters the INITIALIZE state.

DISCONNECTED: The Authenticator enters this state when disassociate or Deauthentication messages is received.

INITIALIZE: The Authenticator enters this state from the DISCONNECTED state, when DeauthenticationRequest event occurs or when the STA initializes. This state initializes the key state variables.

AUTHENTICATION: The Authenticator enters this state when the STA’s management entity sends an AuthentiationRequest to authenticate an SSID.

INITMSK: The authenticator enters this state when the IEEE 802.1X AS signals a successful authentication, or it a pre-shared key is available. If a RadiusKey is supplied it goes to the PTKSTART state, otherwise it goes to the DISCONNECTED state.

> Informative Note. An Authenticator should not allow itself to negotiate IEEE 802.1X if it is not fully configured.

PTKSTART: The Authenticator enters this state from **INITMSK** to start the 4-way handshake, or if no response to the 4-way handshake occurs.

PTKINITNEGOTIATING: The Authenticator enters this state when it receives the second EAPOL-Key message of the 4-way handshake.

UPDATEKEYS: The Authenticator enters this state when it receives an EAPOL-Key message is received from the Supplicant to initiate the 4-way handshake. The key type in the EAPOL-Key message must be set to Pairwise key and the Request bit must be set.

MICFAILURE: The Authenticator enters this state when EAPOL-Key MIC failure occurs—detected either locally, or signaled by peer Supplicant—when the key type indicates a Pairwise key and the Request and Error bits are both set.

REKEYNEGOTIATING: The Authenticator enters this state when a GTK is to be sent to the Supplicant.

> Informative Note: The TxRx flag for sending a Group Key is always the opposite of whether the Pairwise Key is used for data encryption/integrity or not. If a Pairwise key is used for encryption/integrity then the station never transmits with the Group Key otherwise the station uses the Group Key for transmit.

REKEYESTABLISHED: The Authenticator enters this state when it receives an EAPOL-Key message from the supplicant with the key type set to Group key.

KEYERROR: The Authenticator enters this state if the EAPOL-Key acknowledgement for the Group key update is not received before a timeout.

SETKEYS: The Authenticator enters this state when the GTK is to be updated at all Supplicants.

SETKEYSDONE: The Authenticator enters this state when the Group key update has completed.

> Informative Note: SETKEYSDONE calls SetGTK to set the Group key for all associated stations if this fails all communication via this key will fail and the AP needs to detect and recover from this situation.

8.5.6.2 Authenticator state machine variables

AuthenticationRequest – This variable is set TRUE if the STA's IEEE 802.11 Management Entity wants an SSID to be authenticated. This can be set when the STA associates or at other times.

ReAuthenticationRequest – This variable is set TRUE if the IEEE 802.1X Authenticator received an eapStart or 802.1X::reAuthenticate is set.

DeauthenticationRequest – This variable is set TRUE if a disassociation or Deauthentication message is received.

RadiusKeyAvailable – This variable is True is a Radius key was supplied.

EAPOLKeyReceived – This variable is set TRUE when an EAPOL-Key message is received. EAPOL-Key messages that are received in respond to an EAPOL-Key message sent by the Authenticator must contain the same Replay Counter as the Replay Counter in the transmitted message. EAPOL-Key messages that contain different Replay Counters should be discarded. An EAPOL-Key message that is sent by the Supplicant in response to an EAPOL-Key message from the Authenticator must not have the Ack bit set. EAPOL-Key messages sent by the Supplicant not in response to an EAPOL-Key message from the Authenticator must have the Request bit set.

> Informative Note: EAPOL-Key messages with Key Type of Pairwise and a non-zero key index should be ignored.

> Informative Note: EAPOL-Key messages with Key Type of Group and an invalid key index should be ignored.

> Informative Note: When an EAPOL-Key message with the Ack bit not set is received then it is expected as a reply to a message that the Authenticator sent and the replay counter is checked against the replay counter used in the sent EAPOL-Key message. When an EAPOL-Key message with the Request bit set is received then a replay counter for these messages is used, which is a different replay counter than the replay counter used for sending messages to the Supplicant.

TimeoutEvt **-** This variable is set TRUE if the EAPOL_Key packet sent out fails to obtain a response from the Supplicant. The variable may be set by management action, or by the operation of a timeout while in the **PTKSTART** and **REKEYNEGOTIATING** states.

***TimeoutCtr* –** This variable maintains the count of EAPOL-Key receive timeouts. It is incremented each time a timeout occurs on EAPOLKeyRcvd event and is initialized to 0. Clause 8.6.5.3 contains details of the timeout values. The Replay Counter for the EAPOL-Key message shall be incremented on each transmission of the EAPOL-Key message.

L2Failure. – This variable is set if IEEE 802.11 fails to send the EAPOL-Key message containing the Group key to the station.

MICVerified **-** This variable is set to TRUE if the MIC on the received EAPOL Key packet is verified and is correct. Any EAPOL-Key messages with an invalid MIC will be dropped and ignored.

GTKAuthenticator - This is TRUE if the Authenticator is on an AP or it is the designated Authenticator for an IBSS.

IntegrityFailed **-** This is set to TRUE when a data integrity error (i.e. Michael failure) occurs.

> Information Note: This is not the same as MICVerified since IntegrityFailed is generated if the MAC integrity check fails, MICVerified is generated from validating the EAPOL-Key MIC.

GKeyDoneStations - Count of number of stations left to have their Group key updated.

GTKRekey – This variable is set to TRUE when a Group key update is required.

GInitAKeys – This variable is set to TRUE when the Group key update state machine is required.

GInitDone – This variable is set to TRUE when the Group key update state machine has been initialized.

GUpdateStationKeys – This variable is set to TRUE when a new Group key is available to be sent to Supplicants.

GNoStations – This variable counts the number of Authenticators so it is known how many Supplicants need to be sent the Group key.

GkeyReady – This variable is set to TRUE when a Group key has been sent to all current Supplicants. This is used by new Authenticator state machines to decide whether a Group key is available to immediately send to its Supplicant.

PInitAKeys – This variable is set to TRUE when the Authenticator is ready to send a Group key to its Supplicant after initialization.

Counter – This variable is the global station Key Counter used for generating Nonces.

ANonce – This variable holds the current Nonce to be used if the station is an Authenticator.

GNonce – This variable holds the current Nonce to be used if the station is a Group key Authenticator.

*GN***,** *GM* – These are the current key indexes for Group keys. Swap(GM, GN) means that the global key index in GN is swapped with the global key index in GM, so now GM and GN are reversed.

PTK – This variable is the current Pairwise transient key.

GTK[]– This variable is the current Group transient keys for each Group key index.

PMK – PMK is the buffer holding the current Pairwise Master Key.

802.1X::XXX – the IEEE 802.1X state variable *XXX*.

8.5.6.3 Authenticator state machine procedures

STADisconnect() – Execution of this procedure disassociates and deauthenticates the station.

CalcGTK(x). – Calculates the Group Transient Key(GTK) using GNonce as the nonce input to the PRF.

RemoveGTK(x)/**Remove PTK** – Deletes GTK or PTK from encryption/integrity engine.

MIC(x) – Computes a Message Integrity Code over the plaintext data.

CheckMIC(). – Verifies the MIC computed by MIC() function.

Waitupto60() **–** This procedure should stop the Authenticator state machines for all stations at this point if the state machines enter this procedure until 60 seconds have gone by from the last exit from this procedure; i.e. the first time this state machine is entered, it can return immediately. The next time it must stop here until at least 60 seconds from the last time someone has left has gone by. If multiple state machines enter this procedure at the same time then 60 seconds must go by for each state machine to leave this procedure.

8.5.7 Nonce generation (Informative)

All stations contain a global Key Counter which is 256 bits in size. It should be initialized at system boot up time to a fresh cryptographic quality random number. Refer to Annex F.9 on random number generation. When the IEEE 802.1X initializes, it is recommended that IEEE 802.1X set the counter value to:

PRF-256(Random number, “Init Counter”, Local MAC Address || Time)

The Local MAC Address should be AA on the Authenticator and SA on the Supplicant.

Random number should be the best possible random number possible and 256 bits in size. Time should be the current time (from NTP or another time in NTP format) whenever possible. This initialization is to ensure that different initial Key Counter values occur across system restarts whether a real-time clock is available or not. The Key Counter must be incremented (all 256 bits) each time a value is used as a nonce or IV. The Key Counter must not be allowed to wrap to the initialization value, and should be reinitialized using a new random number if this happens.

8.6 Mapping EAPOL keys to 802.11 keys

8.6.1 Mapping PTK to TKIP keys

8.5.1.2 defines the EAPOL temporal keys TK1 and TK2 derived from PTK.

A STA shall use TK1 as its input to the TKIP Phase 1 Mixing Function.

A STA shall use bits 0-63 of TK2 as the Michael key for MSDUs from the Authenticator’s STA to the Supplicant’s STA.

A STA shall use bits 64-127 of TK2 as the Michael key for MSDUs from the STA with the larger MAC address to the STA with the smaller MAC address.

8.6.2 Mapping GTK to TKIP keys

8.5.1.3 defines the EAPOL temporal keys TK1 and TK2 derived from GTK.

A STA shall use TK1 as the input to the TKIP Phase 1 Mixing Function.

A STA shall use bits 0-63 of TK2 as the Michael key for MSDUs from the Authenticator’s STA to the STA .

A STA shall use bits 64-127 of TK2 as the Michael key for MSDUs from the Supplicant’s STA to the Authenticator’s STA.

8.6.3 Mapping PTK to WRAP keys

8.5.1.2 defines the EAPOL temporal keys TK1 and TK2 derived from PTK.

A STA shall use TK1 as the WRAP key for MSDUs between the two communicating STAs.

A STA shall not use TK 2 with WRAP.

8.6.4 Mapping GTK to WRAP keys

8.5.1.3 defines the EAPOL temporal keys TK1 and TK2 derived from GTK.

A STA shall use TK1 as the WRAP key for MSDUs between the two communicating STAs.

A STA shall not use TK2 with WRAP.

8.6.5 Mapping PTK to CCMP keys

8.5.1.2 defines the EAPOL temporal keys TK1 and TK2 derived from PTK.

A STA shall use TK1 as the CCMP key for MSDUs between the two communicating STAs.

A STA shall not use TK 2 with CCMP.

8.6.6 Mapping GTK to CCMP keys

8.5.1.3 defines the EAPOL temporal keys TK1 and TK2 derived from GTK.

A STA shall use TK1 as the CCMP key for MSDUs between the two communicating STAs.

A STA shall not use TK2 with CCMP.

8.6.7 Mapping GTK to WEP-40 keys

8.5.1.3 defines the EAPOL temporal keys TK1 and TK2 derived from GTK.

A STA shall use bits 0-39 of TK1 as the WEP-40 key for MSDUs between the two communicating STAs.

A STA shall not use TK2 with WEP-40.

8.6.8 Mapping GTK to WEP-104 keys

8.5.1.3 defines the EAPOL temporal keys TK1 and TK2 derived from GTK.

A STA shall use bits 0-103 of TK1 as the WEP-104 key for MSDUs between the two communicating STAs.

A STA shall not use TK2 with WEP-104.

8.7 Temporal key processing

Since IEEE 802.1X provides MSDU filtering based on port status, , the 802.11 MAC need not apply filtering except to support legacy WEP behavior in a TSN:

1. dott11PrivacyInvoked shall be true in order for a STA to apply RSN protections.
2. STAs protect all MSDUs when temporal keys are configured, and send and receives all MSDUs in the clear when temporal keys not configured.
3. STAs protect IEEE 802.1X messages only with a key-mapping key; STAs shall not protect IEEE 802.1X messages with default keys.
4. STAs must always be prepared to send or receive IEEE 802.1X data messages in the clear.
5. An AP should disassociate and/or deauthenticate a station on receiving an IEEE 802.1X authFail event for the STA.

8.7.1 Per-MSDU Tx pseudo-code

```
if dot11PrivacyInvoked = FALSE then
        transmit the MSDU without protections
else
        // If we find a suitable unicast or group key for the mode we are in…
        if (MSDU has an individual RA and dot11WEPKeyMappings has an entry for that RA
        and dot11WEPKeyMappingsKeyBroadcast is false) or (the MDPU has a multicast RA
        and the network type is IBSS and the network is RSN and there is an entry in
        dot11KeyMapppings for the TA and dot11WEPKeyMappingsKeyBroadcast is true) then
                if entry has WEPOn = FALSE then
                        transmit the MSDU without protections
                else
                        if that entry contains a null key then
                                discard the entire MSDU and generate an
                                MA-UNITDATA-STATUS.indication primitive to
                                notify LLC that the MSDU was undeliverable due to
                                a null WEP key
                        else
                                // Note that it is assumed that no entry will be in the key
                                // mapping table of a cipher type that is unsupported.
                                Set the KeyID subfield of the IV field to zero.
                                if cipher type of entry is AES-CCM
                                        Transmit the MSDU, to be protected after fragmentation
                                        using AES-CCM and
                                        dot11WEPDefaultKeys[dot11WEPDefaultKeyID]
                                else if cipher type of entry is AES-OCB.
                                        Protect MSDU with AES-OCB cipher and entry's key
                                        Transmit the protected MSDU
                                else if cipher type of entry is TKIP
                                        Compute MIC using Michael algorithm and entry's Tx
                                        MIC key.
                                        Append MIC to MSDU
                                        Transmit the MSDU, to be protected with TKIP and
                                        dot11WEPDefaultKeys[dot11WEPDefaultKeyID]
                                else if cipher type of entry is WEP and dot11RSNEnabled = "false".
                                        Transmit the MSDU, to be protected with WEP and
                                        dot11WEPDefaultKeys[dot11WEPDefaultKeyID]
                                end if
                        endif
```

```
                        endif
                else    // Else we didn't find a key but we are protected, so handle the default key case or discard
                        // But 1st, the following covers the case of an AP in a BSS with encryption, that accepts
                        // non-protected STAs into the BSS and so must transmit broadcasts as plaintext.
                        if MPDU has a group RA and the Privacy subfield of the Capability Information
                        field in this BSS is set to 0 then
                                the MPDU is transmitted without protections
                        else    // No key found so try either default WEP
                                if dot11WEPDefaultKeys[dot11WEPDefaultKeyID] = null then
                                        if Ethertype is 802.1X then
                                                transmit the MPDU without protection
                                        else
                                                discard the MSDU and generate an
                                                MA-UNITDATA-STATUS.indication primitive to
                                                notify LLC that the entire MSDU was undeliverable
                                                due to a null WEP key
                                else if dot11WEPDefaultKeys[dot11WEPDefaultKeyID] is not null
                                        Set the KeyID subfield of the IV field to dot11WEPDefaultKeyID.
                                        if cipher type of entry is AES-CCM
                                                Transmit the MSDU, to be protected after fragmentation
                                                using AES-CCM and
                                                dot11WEPDefaultKeys[dot11WEPDefaultKeyID]
                                        else if cipher type of entry is AES-OCB.
                                                Protect MSDU with AES-OCB cipher and entry's key
                                                Transmit the protected MSDU
                                        else if cipher type of entry is TKIP
                                                Compute MIC using Michael algorithm and entry's Tx
                                                MIC key.
                                                Append MIC to MSDU
                                                Transmit the MSDU, to be protected with TKIP and
                                                dot11WEPDefaultKeys[dot11WEPDefaultKeyID]
                                        else if cipher type of entry is WEP and dot11RSNEnabled = "false".
                                                Transmit the MSDU, to be protected with WEP and
                                                dot11WEPDefaultKeys[dot11WEPDefaultKeyID]
                                        end if
                                endif
                        endif
                endif
        endif
```

8.7.2 Per MPDU Tx pseudo-code

```
        if MPDU is member of an MSDU that is to be transmitted without protections
                transmit the MPDU without protections
        else if MSDU that MPDU is a member of was protected using AES-OCB
                Transmit the MPDU unaltered
        else if MSDU that MPDU is a member of is to be protected using AES-CCM
                Protect the MPDU using entry's key and AES-CCM
                Transmit the MPDU
        else if MSDU that MPDU is a member of is to be protected using TKIP
                Protect the MPDU using TKIP encryption
                Transmit the MPDU
        else if MSDU that MPDU is a member of is to be protected using WEP
                Encrypt the MPDU using entry's key and WEP
                Transmit the MPDU
        else
                // should not arrive here
        endif
```

8.7.3 Per MPDU Rx pseudo-code

```
if the Protected Frame subfield of the Frame Control Field is zero then
    Receive the unencrypted MPDU without protections
else
    if (dot11PrivacyOptionImplemented = "true" and the MPDU has individual RA and there is an
    entry in dot11WEPKeyMappings matching the MPDU's TA and
    dot11WEPKeyMappingsKeyBroadcast is false) or (the MPDU has a multicast RA and the network
    type is IBSS and the network is RSN and there is an entry in dot11KeyMappings for the TA and
    dot11WEPKeyMappingsKeyBroadcast is true) then
        if entry has an AES-OCB key
            receive the frame unaltered
        else if entry has an AES-CCM key
            decrypt frame using AES-CCM key
            discard the frame if the integrity check fails
        else if entry has a TKIP key
            prepare a temporal key from the TA, TKIP key and PN
            decrypt the frame using RC4
            discard the frame if the ICV fails
        else if entry has a WEP key
            decrypt the frame using WEP decryption
            discard the frame if the ICV fails and increment
            dot11WEPUndecryptableCount
        endif
        discard the frame body and increment dot11WEPUndecryptableCount
    else if dot11WEPDefaultKeys[KeyID] is null then
        discard the frame body and increment dot11WEPUndecryptableCount
    else if dot11WEPDefaultKeys[KeyID] is a CCM key
            decrypt and authenticate MPDU using CCMP
        else if dot11WEPDefaultKeys[KeyID] is a WRAP key
            Receive the MPDU, since decryption will take place at MSDU level
        else if dot11WEPDefaultKeys[KeyID] is a TKIP key
            Decrypt the MPDU using TKIP
        else if dot11WEPDefaultKeys[KeyID] is a WEP key
            attempt to decrypt with dot11WEPDefaultKeys[KeyID],
            incrementing dot11WEPICVErrorCount if the ICV check fails
        end if
endif
```

8.7.4 Per MSDU Rx pseudo-code

```
if the frame was not protected
    if MPDU has a group RA and aHaveGTK ="false" and dot11RSNEnabled = "true" then
        Receive the frame unencrypted
    else if aHavePTK = "false" and dot11RSNEnabled = "true"  then //Unicast
        Receive frame unencrypted
    else if dot11RSNEnabled = "false" and aExcludeUnencrypted = "false"
        Receive the frame without applying protections
    else
        discard the frame body without indication to LLC and
        increment dot11WEPExcludedCount
    endif
else            // Have a protected MSDU
    if dot11PrivacyOptionImplemented = TRUE then
        if dot11WEPKeyMappings  entry has WEPOn set to FALSE then
            discard the frame body and increment
            dot11WEPUndecryptableCount
        else if dot11WEPKeyMappings  entry contains a key that is null then
            discard the frame body and increment
            dot11WEPUndecryptableCount
        else if dot11WEPKeyMappings  has an AES-OCB key then
            Decrypt the frame using AES-OCB
```

```
                Discard the frame if authentication fails
                Accept the MSDU of the OCB decryption and authentication succeeds
            else if dot11WEPKeyMappings has an AES-CCM key then
                Accept the MSDU since the decryption took place at the MPDU
            else if dot11WEPKeyMappings has a TKIP key then
                Compute the MIC using the Michael algorithm
                Compare the received MIC against the computed MIC
                discard the frame if the MIC fails and invoke countermeasures if appropriate,
                otherwise accept the MSDU
            else if dot11WEPKeyMappings has a WEP key then
                Accept the MSDU since the decryption took place at the MPDU
            endif
        else // Cannot decrypt payload, so discard it.
            discard the frame body
        endif
    endif
```

[Editorial note: end of Clause 8]

In clause 10.3.11.1.2:

Rename SharedID to KeyID

Change description for SharedID to

This parameter is valid only when the Use of the Key includes ENCRYPT. The KeyID to be assigned to this Key.

10.3.2.2.2 Semantics of the service primitive

Add the following rows at the end of the BSSDescription in Clause 10.3.2.2.2:

RSN Information Element	RSN Information Element	As defined in frame format.	A description of the cipher suites and authenticated key management suites supported in the BSS.

10.3.6.1.2 Semantics of the service primitive

Add the following parameters to the MLME-ASSOCIATE.request primitive in Clause 10.3.6.1.2:

Authenticated Key Management selector,
Pairwise Key Cipher Suite selector

Add the following rows at the end of the table in Clause 10.3.6.1.2 defining the MLME-ASSOCIATE.request:

Authenticated Key	Integer	As defined in RSN IE	Authenticated Key Management

Management selector		format	Suite requested for this association
Pairwise Key Cipher Suite selector	Integer	As defined in RSN IE format	The Pairwise Key Cipher Suite requested for this association

10.3.6.3.2 Semantics of the service primitive

Add the following parameters to the MLME-ASSOCIATE.indication primitive in Clause 10.3.6.3.2:

Authenticated Key Management selector,
Pairwise Key Cipher Suite selector

Add the following rows at the end of the table in Clause 10.3.6.3.2 defining the MLME-ASSOCIATE.indication:

Authenticated Key Management selector	Integer	As defined in RSN IE format	The Authenticated Key Management Suite requested for this association
Pairwise Key Cipher Suite selector	Integer	As defined in RSN IE format	The Pairwise Key Cipher Suite requested for this association

10.3.7.1.2 Semantics of the service primitive

Add the following parameters to the MLME-REASSOCIATE.request primitive in Clause 10.3.7.1.2:

Authenticated Key Management selector,
Pairwise Key Cipher Suite selector

Add the following rows at the end of the table in Clause 10.3.7.1.2 defining the MLME-REASSOCIATE.request:

Authenticated Key Management selector	Integer	As defined in RSN IE format	The Authenticated Key Management Suite requested for this association
Pairwise Key Cipher Suite selector	Integer	As defined in RSN IE format	The Pairwise Key Cipher Suite requested for this association

10.3.7.3.2 Semantics of the service primitive

Add the following parameters to the MLME-REASSOCIATE.indication primitive in Clause 10.3.7.3.2:

Authenticated Key Management selector,
Pairwise Key Cipher Suite selector

Add the following rows at the end of the table in Clause 10.3.7.3.2 defining the MLME-REASSOCIATE.indication:

Authenticated Key Management selector	Integer	As defined in RSN IE format	The Authenticated Key Management Suite requested for this association
Pairwise Key Cipher Suite selector	Integer	As defined in RSN IE format	The Pairwise Key Cipher Suite requested for this association

10.3.8.1.2 Semantics of the service primitive

Add the following Clauses after Clause 10.3.10.2.4, but prior to Clause 10.4, renumbering as appropriate:

10.3.11 SetKeys

10.3.11.1 MLME-SETKEYS.request

10.3.11.1.1 Function

This primitive causes the keys identified in the parameters of the primitive to be set in the MAC and enabled for use.

10.3.11.1.2 Semantics of the Service Primitive

The primitive parameters are as follows:

```
MLME-SETKEYS.request        (
                            Keylist
                            )
```

Name	Type	Valid range	Description
Keylist	A set of KeyIdentifiers	N/A	The list of keys to be used by the MAC.

Each KeyIdentifier consists of the following elements:

Name	Type	Valid range	Description
Key	Bit string	N/A	The key value
Length	Integer	N/A	The number of bits in the Key to be used.
Index	Integer	N/A	Key Index
Type	Integer	Group, Pairwise	Defines whether this key is a Group or Pairwise key.

Tx	Boolean	TRUE, FALSE	This parameter indicates if this key is to be used for transmission and reception or just reception.
Address	MAC Address	Any valid individual MAC address	This parameter is valid only when the key type is Pairwise and contains an IEEE 802 address
Receive Sequence Count	8 octets	N/A	Value the receive sequence counter should be initialized to
Authenticator/Supplicant	Boolean	TRUE, FALSE	Whether the key is set by the Authenticator or Supplicant. IEEE 802.11 uses this to select the correct integrity key when Michael is used.

10.3.11.1.3 When Generated

This primitive is generated by the SME at any time when one or more keys are to be set in the MAC.

10.3.11.1.4 Effect of Receipt

Receipt of this primitive causes the MAC to set the appropriate keys and to begin using them as indicated. If the AES-based privacy algorithm is being used for unicast traffic over this association, the MAC derives the keys as specified in 8.3.2.3.4.

10.3.11.2 MLME-SETKEYS.confirm

10.3.11.2.1 Function

This primitive confirms that the action of the associated MLME-SETKEYS.request has been completed.

10.3.11.2.2 Semantics of the service primitive

This primitive has no parameters.

10.3.11.2.3 When Generated

This primitive is generated by the MAC in response to receipt of a MLME-SETKEYS.request primitive. This primitive is issued when the action requested has been completed.

10.3.11.2.4 Effect of Receipt

The SME is notified that the requested action of the MLME-SETKEYS.request is completed.

10.3.12 DeleteKeys

10.3.12.1 MLME-DELETEKEYS.request

10.3.12.1.1 Function

This primitive causes the keys identified in the parameters of the primitive to be deleted from the MAC and thus disabled for use.

10.3.12.1.2 Semantics of the Service Primitive

The primitive parameters are as follows:

MLME-DELETEKEYS.request (
Keylist,
)

Name	Type	Valid range	Description
Keylist	A set of KeyIdentifiers	N/A	The list of keys to be deleted from the MAC

Each KeyIdentifier consists of the following elements:

Name	Type	Valid range	Description
Address	MAC Address	Any valid individual MAC address	This parameter is valid only when the key type is Pairwise and contains an 802 address

10.3.11.1.3 When Generated

This primitive is generated by the SME at any time when keys for a security association are to be deleted in the MAC.

10.3.11.1.4 Effect of Receipt

Receipt of this primitive causes the MAC to

1. Delete the appropriate keys, both group and pairwise and to cease using them.
2. Set aHaveGTK to FALSE
3. Set aHavePTK to FALSE

10.3.12.2 MLME-DELETEKEYS.confirm

10.3.12.2.1 Function

This primitive confirms that the action of the associated MLME-DELETEKEYS.request has been completed.

10.3.12.2.2 Semantics of the service primitive

This primitive has no parameters.

10.3.12.2.3 When Generated

This primitive is generated by the MAC in response to receipt of a MLME-DELETEKEYS.request primitive. This primitive is issued when the action requested has been completed.

10.3.12.2.4 Effect of Receipt

The SME is notified that the requested action of the MLME-DELETEKEYS.request is completed.

Insert the following clause:

11.3.1 Stations association procedures

Change the text of Clause 11.3.1 from:

Upon receipt of an MLME-ASSOCIATE.request, a STA shall associate with an AP via the following procedure:

a) The STA shall transmit an association request to an AP with which that STA is authenticated.

b) If an Association Response frame is received with a status value of “successful,” the STA is now associated with the AP and the MLME shall issue an MLME-ASSOCIATE.confirm indicating the successful completion of the operation.

c) If an Association Response frame is received with a status value other than “successful” or the AssociateFailureTimeout expires, the STA is not associated with the AP and the MLME shall issue an MLME-ASSOCIATE.confirm indicating the failure of the operation.

to:

Upon receipt of an MLME-ASSOCIATE.request, a STA shall associate with an AP via the following procedure:

a) The STA shall transmit an association request to an AP with which that STA is authenticated. If the STA is operating in an RSN, the STA shall include the RSN IE with only one pairwise key cipher suite and only one authenticated key suite.

b) If an Association Response frame is received with a status value of “successful,” the STA is now associated with the AP and the MLME shall issue an MLME-ASSOCIATE.confirm indicating the successful completion of the operation.

c) If an Association Response frame is received with a status value other than “successful” or the AssociateFailureTimeout expires, the STA is not associated with the AP and the MLME shall issue an MLME-ASSOCIATE.confirm indicating the failure of the operation.

11.3.2 AP association procedures

Change the text of Clause 11.3.2 from:

An AP shall operate as follows in order to support the association of STAs.

a) Whenever an Association Request frame is received from a STA and the STA is authenticated, the AP shall transmit an association response with a status code as defined in 7.3.1.9. If the status value is “successful,” the Association ID assigned to the STA shall be included in the response. If the STA is not authenticated, the AP shall transmit a Deauthentication frame to the STA.

b) When the association response with a status value of “successful” is acknowledged by the STA, the STA is considered to be associated with this AP.

c) The AP shall inform the distribution system (DS) of the association and the MLME shall issue an MLME-ASSOCIATE.indication.

to:

An AP shall operate as follows in order to support the association of STAs.

a) Whenever an Association Request frame is received from a STA and the STA is authenticated, the AP shall transmit an association response with a status code as defined in 7.3.1.9. If the AP is operating as an RSN, the AP will check the values received in the RSN IE, to see if the values received match the APs security policy. If the status value is "successful," the Association ID assigned to the STA shall be included in the response. If the STA is not authenticated, the AP shall transmit a Deauthentication frame to the STA.

b) When the association response with a status value of "successful" is acknowledged by the STA, the STA is considered to be associated with this AP.

c) The AP shall inform the distribution system (DS) of the association and the MLME shall issue an MLME-ASSOCIATE.indication.

11.3.4 AP Reassociation procedures

Change the text of Clause 11.3.4 from:

An AP shall operate as follows in order to support the Reassociation of STAs.

a) Whenever a Reassociation Request frame is received from a STA and the STA is authenticated, the AP shall transmit a Reassociation response with a status value as defined in 7.3.1.9. If the status value is "successful," the Association ID assigned to the STA shall be included in the response. If the STA is not authenticated, the AP shall transmit a Deauthentication frame to the STA.

b) When the Reassociation response with a status value of "successful" is acknowledged by the STA, the STA is considered to be associated with this AP.

c) The AP shall inform the DS of the Reassociation and the MLME shall issue an MLME-REASSOCIATE. indication.

to:

An AP shall operate as follows in order to support the Reassociation of STAs.

a) Whenever a Reassociation Request frame is received from a STA and the STA is authenticated, the AP shall transmit a Reassociation response with a status value as defined in 7.3.1.9. If the AP is operating as an RSN, the AP will check the values received in the RSN IE, to see if the values received match the APs security policy. If the status value is "successful," the Association ID assigned to the STA shall be included in the response. If the STA is not authenticated, the AP shall transmit a Deauthentication frame to the STA.

b) When the Reassociation response with a status value of "successful" is acknowledged by the STA, the STA is considered to be associated with this AP.

c) The AP shall inform the DS of the Reassociation and the MLME shall issue an MLME-REASSOCIATE. indication.

Annex A (normative) Protocol Implementation Conformance Statements (PICS)

Add the following text to this annex, where "X" in PCX is the next number for the protocol capabilities:

Item	Protocol Capability	References	Status	Support
	Are the following MAC protocol capabilities supported?			
PCX	Robust Security Network		O	Yes o No o
PCX.1	RSN IE	7.3.2.17	PCX:M, FT1:M, FR1:M, FT3:M, FR3:M, FT6:M, FR6:M, FT7:M, FR7:M	Yes o No o
PCX.1.1	Group Key Cipher Suite	7.3.2.17	PCX.1:M	Yes o No o
PCX.1.2	Pairwise Key Cipher Suite List	7.3.2.17	PCX.1:M	Yes o No o
PCX.1.2.1	CCMP data privacy protocol	8.3.4	PCX:M	Yes o No o
PCX.1.2.1.1	CCMP encapsulation procedure	8.3.4.1.1	PCX.1.2.1:M	Yes o No o
PCX.1.2.1.2	CCMP decapsulation procedure	8.3.4.1.2	PCX.1.2.1:M	Yes o No o
PCX.1.2.1.3	CCMP Security Serv. Mng.		M	Yes o No o
PCX.1.2.2	TKIP data privacy protocol	8.3.2	O	Yes o No o
PCX.1.2.2.1	TKIP encapsulation procedure	8.3.2.1.1	PCX.1.2.2:M	Yes o No o
PCX.1.2.2.2	TKIP decapsulation procedure	8.3.2.1.2	PCX.1.2.2:M	Yes o No o
PCX.1.2.2.3	TKIP counter measures	8.3.2.4.2	PCX.1.2.2:M	Yes o No o
PCX.1.2.2.4	TKIP Security Serv. Mng.		M	Yes o No o
PCX.1.2.3	WRAP data privacy protocol	8.3.3	O	Yes o No o
PCX.1.2.3.1	WRAP encapsulation procedure	8.3.3.1.1	PCX.1.2.3:M	Yes o No o
PCX.1.2.3.2	WRAP decapsulation procedure	8.3.3.1.2	PCX.1.2.3:M	Yes o No o
PCX.1.2.3.3	WRAP Security Serv. Mng.		M	Yes o No o
PCX.1.3	Auth. Key Mng. Suite List	7.3.2.17	PCX.1:M	Yes o No o
PCX.1.3.1	Unspec. EAP/802.11i Key Mng.	7.3.2.17	PCX.1:M	Yes o No o
PCX.1.3.2	Preshard key/802.11i Key Mng.	7.3.2.17	PCX.1:M	Yes o No o
PCX.1.3.3	802.11i Key Mng.	8.5	PCX.1:M	Yes o No o
PCX.1.3.3.1	Key Hierarchy	8.5	PCX.1:M	Yes o No o
PCX.1.3.3.1.1	Pairwise Key Hierarchy	8.5.1.2	PCX.1:M	Yes o No o
PCX.1.3.3.1.2	Group Key Hierarchy	8.5.1.3	PCX.1:M	Yes o No o
PCX.1.3.3.2	4 way handshake	8.5.3	PCX.1:M	Yes o No o
PCX.1.3.3.3	Group key handshake	8.5.4	PCX.1:M	Yes o No o
PCX.1.4	RSN Capabilities	7.3.2.17	PCX.1:M	Yes o No o

End of annex A text changes

Annex C (normative) Formal description of MAC operation

Delete the text of this annex.

Annex D (normative) ASN.1 encoding of the MAC and PHY MIB

Update following MIB entries in Annex D:

Add the following attribute to the dot11StationConfigTable in Annex D:

```
dot11RSNOptionImplemented OBJECT-TYPE
      SYNTAX       TruthValue
      MAX-ACCESS   read-only
      STATUS       current
      DESCRIPTION
            "This variable indicates whether the entity is RSN-capable."
      ::= { dot11StationConfigEntry 24 }
```

Add the following attribute to the dot11PrivacyTable in Annex D:

```
dot11RSNEnabled OBJECT-TYPE
      SYNTAX       TruthValue
      MAX-ACCESS   read-write
      STATUS       current
      DESCRIPTION
            "When this object is set to TRUE, this shall indicate that
            RSN is enabled on this entity. The entity will advertise the
            RSN Information Element in its Beacons and Probe Responses.
            Configuration variables for RSN operation  are found in the
            dot11RSNConfigTable.

            This object requires that dot11PrivacyInvoked also be set to
            TRUE. "
      ::= { dot11PrivacyEntry 7 }
```

Change the DESCRIPTION clause of object dot11PrivacyInvoked in Annex D from:

```
"When this attribute is true, it shall indicate that the IEEE 802.11 WEP
mechanism is used for transmitting frames of type Data. The default value
of this attribute shall be false."
```

to:

```
"When this attribute is TRUE, it shall indicate that some level of
security is invoked for transmitting frames of type Data. For  802.11-
1999 clients, the security mechanism used is WEP.

For RSN-capable clients, an additional variable dot11RSNEnabled indicates
whether RSN is enabled. If dot11RSNEnabled is FALSE, the security
mechanism invoked  is WEP; if dot11RSNEnabled is TRUE, RSN security
mechanisms invoked are configured in the dot11RSNConfigTable. The default
value of this attribute shall be FALSE. "
```

Add ***to dot11StationConfigEntry***

```
dot11TKIPNumberOfReplayCounters   Integer
```

Add ***definition of dot11TKIPNumberOfReplayCounters***

```
dot11TKIPNumberOfReplayCounters
      SYNTAX INTEGER
      MAX-ACCESS read-only
      STATUS current
      DESCRIPTION
            "Specifies the number of replay counters: 0 – 1 replay
            counter, 1 – 2 replay counters, 2 – 4 replay counters, 3 –
            16 replay counters"
      ::= { dot11StationConfigEntry 2 }
```

Incorporate the following text as the IEEE 802.11i MIB (in the correct Annex: D)

```
--
--  IEEE 802.11i MIB
--
```

```
IEEE802dot11i-MIB DEFINITIONS ::= BEGIN
      IMPORTS
            MODULE-IDENTITY, OBJECT-TYPE, Integer32, Unsigned32,
            Counter32
                  FROM SNMPv2-SMI

            DisplayString, MacAddress, TruthValue
                  FROM SNMPv2-TC

            ieee802dot11
                  FROM IEEE802dot11-MIB

            InterfaceIndexOrZero
                  FROM IF-MIB;

ieee802dot11i MODULE-IDENTITY
      LAST-UPDATED "0209100000Z"
      ORGANIZATION "IEEE 802.11"
      CONTACT-INFO
            "WG E-mail: stds-802-11@ieee.org

      Chair:        Stuart J. Kerry
      Postal:       Philips Semiconductors, Inc.
                    1109 McKay Drive
                    M/S 48 SJ
                    San Jose, CA 95130-1706  USA
      Tel:          +1 408 474 7356
      Fax:          +1 408 474 7247
      E-mail:       stuart.kerry@philips.com

      TGi Chair:    David Halasz
      Postal:
      Tel:
      Fax:
      E-mail:       dhala@cisco.com

      Technical Editor:   Jesse R. Walker
      Postal:       Intel Corporation
                    JF3-466
                    2111 N.E. 25th Avenue
                    Hillsboro, OR 97124-5961 USA
      Tel:          +1 503 712 1849
      Fax:
      Email:        jesse.walker@intel.com
      "

DESCRIPTION
      "The MIB module for 802.11 entities implementing 802.11i
      (RSN/TSN)."
::= { ieee802dot11 7 }

--
--  Robust Security Network (RSN (and TSN)) Configuration
--

dot11RSNConfigTable OBJECT-TYPE
      SYNTAX        SEQUENCE OF Dot11RSNConfigEntry
      MAX-ACCESS    not-accessible
      STATUS        current
      DESCRIPTION
            "The table containing RSN/TSN configuration objects."
      ::= { ieee802dot11i 1 }

dot11RSNConfigEntry OBJECT-TYPE
SYNTAX        Dot11RSNConfigEntry
      MAX-ACCESS    not-accessible
      STATUS        current
      DESCRIPTION
            "An entry in the dot11RSNConfigTable."
      INDEX { dot11RSNConfigIndex }
```

```
                ::= { dot11RSNConfigTable 1 }

        Dot11RSNConfigEntry ::=
                SEQUENCE {
                        dot11RSNConfigIndex              InterfaceIndexOrZero,
                        dot11RSNConfigVersion                   Integer32,
                        dot11RSNConfigPairwiseKeysSupported     Unsigned32,
                        dot11RSNConfigMulticastCipher           OCTET STRING,
                        dot11RSNConfigGroupRekeyMethod          INTEGER,
                        dot11RSNConfigGroupRekeyTime            Unsigned32,
                        dot11RSNConfigGroupRekeyPackets         Unsigned32,
                        dot11RSNConfigGroupRekeyStrict          TruthValue,
                        dot11RSNConfigPSKValue                  OCTET STRING,
                        dot11RSNConfigPSKPassPhrase             DisplayString,
                        dot11RSNConfigTSNEnabled                TruthValue,
                        dot11RSNConfigGroupMasterRekeyTime      Unsigned32,
                        dot11RSNConfigGroupUpdateTimeOut        Unsigned32,
                        dot11RSNConfigGroupUpdateCount          Unsigned32,
                        dot11RSNConfigPairwiseUpdateTimeOut     Unsigned32,
                        dot11RSNConfigPairwiseUpdateCount       Unsigned32
                }

        dot11RSNConfigIndex OBJECT-TYPE
                SYNTAX          InterfaceIndexOrZero
                MAX-ACCESS      not-accessible
                STATUS          current
                DESCRIPTION
                        "Each 802.11 interface is represented by an entry in the
                        ifTable. If this index is zero, the information in this
                        table shall apply to all 802.11 interfaces."
                ::= { dot11RSNConfigEntry 1 }

        dot11RSNConfigVersion OBJECT-TYPE
                SYNTAX          Integer32
                MAX-ACCESS      read-only
                STATUS          current
                DESCRIPTION
                        "The highest RSN version this entity supports."
                ::= { dot11RSNConfigEntry 2 }

        dot11RSNConfigPairwiseKeysSupported OBJECT-TYPE
                SYNTAX          Unsigned32 (0..4294967295)
                MAX-ACCESS      read-only
                STATUS          current
                DESCRIPTION
                        "This object indicates how many pairwise keys the entity
                        supports for RSN. When zero, it only supports (four) default
                        keys."
                ::= { dot11RSNConfigEntry 3 }

        dot11RSNConfigMulticastCipher OBJECT-TYPE
                SYNTAX          OCTET STING (SIZE(4))
                MAX-ACCESS      read-write
                STATUS          current
                DESCRIPTION
                        "This object indicates the multicast cipher suite selector
                        the entity must use. The multicast cipher suite in the RSN
                        Information Element shall take its value from this variable.
                        It consists of an OUI (the three most significant octets)
                        and a cipher suite identifier (the least significant octet).

                        The network administrator can always override the
                        automatically selected multicast cipher suite by writing
                        this object."
                ::= { dot11RSNConfigEntry 4 }

        dot11RSNConfigGroupRekeyMethod OBJECT-TYPE
                SYNTAX          INTEGER { disabled(1), timeBased(2), packetBased(3) }
                MAX-ACCESS      read-write
```

```
        STATUS       current
        DESCRIPTION
              "This object selects a mechanism for rekeying the RSN Group
              Key. The default is time-based, once per day. Rekeying the
              Group key is only applicable to an entity acting in the
              Authenticator role (an AP in an ESS)."
        DEFVAL       { timeBased }
        ::= { dot11RSNConfigEntry 5 }

  dot11RSNConfigGroupRekeyTime OBJECT-TYPE
        SYNTAX       Unsigned32 (1..4294967295)
        UNITS        "seconds"
        MAX-ACCESS   read-write
        STATUS       current
        DESCRIPTION
              "The time in seconds after which the RSN group key must be
              refreshed. The timer shall start at the moment the group key
              was set using the MLME-SetKeys primitive.

              The fine granularity (seconds) also enables the network
              Administrator to ‘immediately’ refresh the group key."
        DEFVAL       { 86400 } -- once per day
        ::= { dot11RSNConfigEntry 6 }

  dot11RSNConfigGroupRekeyPackets OBJECT-TYPE
        SYNTAX       Unsigned32 (1..4294967295)
        UNITS        "1000 packets"
        MAX-ACCESS   read-write
        STATUS       current
        DESCRIPTION
              "A packet count (in 1000s of packets) after which the RSN
              group key shall be refreshed. The packet counter shall start
              at the moment the group key was set using the MLME-SetKeys
              primitive and it shall count all packets encrypted using the
              current group key."
        ::= { dot11RSNConfigEntry 7 }

  dot11RSNConfigGroupRekeyStrict OBJECT-TYPE
        SYNTAX       TruthValue
        MAX-ACCESS   read-write
        STATUS       current
        DESCRIPTION
              "This object signals that the group key shall be refreshed
              whenever a Station leaves the BSS."
        ::= { dot11RSNConfigEntry 8 }

  dot11RSNConfigPSKValue OBJECT-TYPE
        SYNTAX       OCTET STRING (SIZE(32))
        MAX-ACCESS   read-write
        STATUS       current
        DESCRIPTION
              "The Pre-Shared Key (PSK) for when RSN in PSK mode is the
              selected authentication suite. In that case, the PMK will
              obtain its value from this object.

              This object is logically write-only. Reading this variable
              shall return unsuccessful status or null or zero."
        ::= { dot11RSNConfigEntry 9 }

  dot11RSNConfigPSKPassPhrase OBJECT-TYPE
        SYNTAX       DisplayString
        MAX-ACCESS   read-write
        STATUS       current
        DESCRIPTION
              "The PSK, for when RSN in PSK mode is the selected
              authentication suite, is configured by
              dot11RSNConfigPSKValue.

              An alternative manner of setting the PSK uses the password-
              to-key algorithm defined in section XXX. This variable
```

```
                    provides a means to enter a pass phrase. When this object is
                    written, the RSN entity shall use the password-to-key
                    algorithm specified in section XXX to derive a pre-shared
                    and populate dot11RSNConfigPSKValue with this key.

                    This object is logically write-only. Reading this variable
                    shall return unsuccessful status or null or zero."
             ::= { dot11RSNConfigEntry 10 }

      dot11RSNConfigTSNEnabled OBJECT-TYPE
             SYNTAX        TruthValue
             MAX-ACCESS    read-write
             STATUS        current
             DESCRIPTION
                    "When dot11PrivacyInvoked and dot11RSNEnabled are both set
                    to TRUE, signaling that RSN is enabled on this entity, this
                    object shall indicate the entity also supports pre-RSN
                    clients (with or without an IEEE 802.1X supplicant), also
                    referred to as a Transitional Security Network (TSN)."
             ::= { dot11RSNConfigEntry 11 }

      dot11RSNConfigGroupMasterRekeyTime OBJECT-TYPE
             SYNTAX        Unsigned32 (1..4294967295)
             UNITS         "seconds"
             MAX-ACCESS    read-write
             STATUS        current
             DESCRIPTION
                    "The time in seconds after which the RSN group master key
                    must be changed. The timer shall start at the moment the
                    group master key was set.

                    A group key refresh will occur on a group master key change.

                    The fine granularity (seconds) also enables the network
                    Administrator to 'immediately' refresh the group master
                    key."
             DEFVAL        { 7604800 } – 604800 = 7*86400, once per week
             ::= { dot11RSNConfigEntry 12 }

      dot11RSNConfigGroupUpdateTimeOut OBJECT-TYPE
             SYNTAX        Unsigned32 (1..4294967295)
             UNITS         "mili-seconds"
             MAX-ACCESS    read-write
             STATUS        current
             DESCRIPTION
                    "The time in mili-seconds after which the RSN group update
                    handshake will be retried. The timer shall start at the
                    moment the group update message is sent."
             DEFVAL        { 100 } --
             ::= { dot11RSNConfigEntry 13 }

      dot11RSNConfigGroupUpdateCount OBJECT-TYPE
             SYNTAX        Unsigned32 (1..4294967295)
             MAX-ACCESS    read-write
             STATUS        current
             DESCRIPTION
                    "The number of times the RSN Group update will be retried."
             DEFVAL        { 3 } --
             ::= { dot11RSNConfigEntry 14 }

      dot11RSNConfigPairwiseUpdateTimeOut OBJECT-TYPE
             SYNTAX        Unsigned32 (1..4294967295)
             UNITS         "mili-seconds"
             MAX-ACCESS    read-write
             STATUS        current
             DESCRIPTION
                    "The time in mili-seconds after which the RSN 4-way
                    handshake will be retried. The timer shall start at the
                    moment a 4-way message is sent."
             DEFVAL        { 100 } --
```

```
        ::= { dot11RSNConfigEntry 15 }

dot11RSNConfigPairwiseUpdateCount OBJECT-TYPE
        SYNTAX       Unsigned32 (1..4294967295)
        MAX-ACCESS   read-write
        STATUS       current
        DESCRIPTION
                "The number of times the RSN 4-way handshake will be
                retried."
        DEFVAL       { 3 } --
        ::= { dot11RSNConfigEntry 16 }

--
--  Unicast Cipher Suite configuration table
--

dot11RSNConfigUnicastCiphersTable OBJECT-TYPE
        SYNTAX       SEQUENCE OF Dot11RSNConfigUnicastCiphersEntry
        MAX-ACCESS   not-accessible
        STATUS       current
        DESCRIPTION
                "This table lists the unicast ciphers supported by this
                entity. It allows enabling and disabling of each unicast
                cipher by network management. The Unicast Cipher Suite list
                in the RSN Information Element is formed using the
                information in this table."
        ::= { ieee802dot11i 2 }

dot11RSNConfigUnicastCiphersEntry OBJECT-TYPE
        SYNTAX       Dot11RSNConfigUnicastCiphersEntry
        MAX-ACCESS   not-accessible
        STATUS       current
        DESCRIPTION
                "The table entry, indexed by the interface index (or all
                interfaces) and the unicast cipher."
        INDEX { dot11RSNConfigIndex, dot11RSNConfigUnicastCipherIndex }
        ::= { dot11RSNConfigUnicastCiphersTable 1 }

Dot11RSNConfigUnicastCiphersEntry ::=
        SEQUENCE {
                dot11RSNConfigUnicastCipherIndex Unsigned32,
                dot11RSNConfigUnicastCipher             OCTET STRING,
                dot11RSNConfigUnicastCipherEnabled      TruthValue  }

dot11RSNConfigUnicastCipherIndex OBJECT-TYPE
        SYNTAX       Unsigned32 (1..4294967295)
        MAX-ACCESS   not-accessible
        STATUS       current
        DESCRIPTION
                "The auxiliary index into the
                dot11RSNConfigUnicastCiphersTable."
        ::= { dot11RSNConfigUnicastCiphersEntry 1 }

dot11RSNConfigUnicastCipher OBJECT-TYPE
        SYNTAX       OCTET STRING (SIZE(4))
        MAX-ACCESS   read-only
        STATUS       current
        DESCRIPTION
                "The selector of a supported unicast cipher. It consists of
                an OUI (the three most significant octets) and a cipher
                suite identifier (the least significant octet)."
        ::= { dot11RSNConfigUnicastCiphersEntry 2 }

dot11RSNConfigUnicastCipherEnabled OBJECT-TYPE
        SYNTAX       TruthValue
        MAX-ACCESS   read-write
        STATUS       current
        DESCRIPTION
                "This object enables or disables the unicast cipher."
        ::= { dot11RSNConfigUnicastCiphersEntry 3 }
```

```
--
--  The Authentication Suites Table
--
dot11RSNConfigAuthenticationSuitesTable OBJECT-TYPE
      SYNTAX       SEQUENCE OF Dot11RSNConfigAuthenticationSuitesEntry
      MAX-ACCESS   not-accessible
      STATUS       current
      DESCRIPTION
            "This table lists the authentication suites supported by
            this entity. Each authentication suite can be individually
            enabled and disabled. The Authentication Suite List in the
            RSN IE is formed using the information in this table."
      ::= { ieee802dot11i 3 }

dot11RSNConfigAuthenticationSuitesEntry OBJECT-TYPE
      SYNTAX       Dot11RSNConfigAuthenticationSuitesEntry
      MAX-ACCESS   not-accessible
      STATUS       current
      DESCRIPTION
            "An entry (row) in the
            dot11RSNConfigAuthenticationSuitesTable."
      INDEX { dot11RSNConfigAuthenticationSuiteIndex }
      ::= { dot11RSNConfigAuthenticationSuitesTable 1 }

Dot11RSNConfigAuthenticationSuitesEntry ::=
      SEQUENCE {
            dot11RSNConfigAuthenticationSuiteIndex         Unsigned32,
            dot11RSNConfigAuthenticationSuite       OCTET STRING,
            dot11RSNConfigAuthenticationSuiteEnabled       TruthValue }

dot11RSNConfigAuthenticationSuiteIndex OBJECT-TYPE
      SYNTAX       Unsigned32 (1..4294967295)
      MAX-ACCESS   not-accessible
      STATUS       current
      DESCRIPTION
            "The auxiliary variable used as an index into the
            dot11RSNConfigAuthenticationSuitesTable."
      ::= { dot11RSNConfigAuthenticationSuitesEntry 1 }

dot11RSNConfigAuthenticationSuite OBJECT-TYPE
      SYNTAX       OCTET STRING (SIZE(4))
      MAX-ACCESS   read-only
      STATUS       current
      DESCRIPTION
            "The selector of an authentication suite. It consists of an
            OUI (the three most significant octets) and a cipher suite
            identifier (the least significant octet). "
      ::= { dot11RSNConfigAuthenticationSuitesEntry 2 }

dot11RSNConfigAuthenticationSuiteEnabled OBJECT-TYPE
      SYNTAX       TruthValue
      MAX-ACCESS   read-write
      STATUS       current
      DESCRIPTION
            "This variable indicates whether the corresponding
            authentication suite is enabled/disabled."
      ::= { dot11RSNConfigAuthenticationSuitesEntry 3 }

--
--  RSN/TSN statistics
--

dot11RSNStatsTable OBJECT-TYPE
      SYNTAX       SEQUENCE OF Dot11RSNStatsEntry
      MAX-ACCESS   not-accessible
      STATUS       current
      DESCRIPTION
            "This table maintains per-STA statistics for SN. The entry
            with dot11RSNStatsSTAAddress set to FF-FF-FF-FF-FF-FF shall
            contain statistics for broadcast/multicast traffic."
```

```
        ::= { ieee802dot11i 4 }

dot11RSNStatsEntry OBJECT-TYPE
        SYNTAX        Dot11RSNStatsEntry
        MAX-ACCESS    not-accessible
        STATUS        current
        DESCRIPTION
                "An entry in the dot11RSNStatsTable."
        INDEX { dot11RSNConfigIndex, dot11RSNStatsIndex }
        ::= { dot11RSNStatsTable 1 }

Dot11RSNStatsEntry ::=
        SEQUENCE {
                dot11RSNStatsIndex                       Unsigned32,
                dot11RSNStatsSTAAddress                  MacAddress,
                dot11RSNStatsVersion                     Unsigned32,
                dot11RSNStatsSelectedUnicastCipher       OCTET STRING,
                dot11RSNStatsTKIPICVErrors               Counter32,
                dot11RSNStatsTKIPLocalMICFailures        Counter32,
                dot11RSNStatsTKIPRemoteMICFailures       Counter32,
                dot11RSNStatsTKIPCounterMeasuresInvoked Counter32,
                dot11RSNStatsWRAPFormatErrors            Counter32,
                dot11RSNStatsWRAPReplays                 Counter32,
                dot11RSNStatsWRAPDecryptErrors           Counter32,
                dot11RSNStatsCCMPFormatErrors            Counter32,
                dot11RSNStatsCCMPReplays                 Counter32,
                dot11RSNStatsCCMPDecryptErrors           Counter32  }

dot11RSNStatsIndex OBJECT-TYPE
        SYNTAX        Unsigned32 (1..4294967295)
        MAX-ACCESS    not-accessible
        STATUS        current
        DESCRIPTION
                "An auxiliary index into the dot11RSNStatsTable."
        ::= { dot11RSNStatsEntry 1 }

dot11RSNStatsSTAAddress OBJECT-TYPE
        SYNTAX        MacAddress
        MAX-ACCESS    read-only
        STATUS        current
        DESCRIPTION
                "The MAC address of the station the statistics in this
                conceptual row belong to."
        ::= { dot11RSNStatsEntry 2 }

dot11RSNStatsVersion OBJECT-TYPE
        SYNTAX        Unsigned32 (1..4294967295)
        MAX-ACCESS    read-only
        STATUS        current
        DESCRIPTION
                "The RSN version which the station associated with."
        ::= { dot11RSNStatsEntry 3 }

dot11RSNStatsSelectedUnicastCipher OBJECT-TYPE
        SYNTAX        OCTET STRING (SIZE(4))
        MAX-ACCESS    read-only
        STATUS        current
        DESCRIPTION
                "The Authentication Suite the station selected during
                association. The value consists of a three octet OUI
                followed by a one octet Type as follows:

        OUI      Value Authentication Type           Key Management Type
        -------- ----- -------------------           -------------------
        00:00:00 0     Reserved                      Reserved
        00:00:00 1     Unspecified authentication    802.1X Key Management
                                                     over 802.1X
        00:00:00 2     None                          802.1X Key Management
                                                     using pre-shared Key
```

```
        00:00:00 3-255 Reserved                    Reserved
        Vendor   any   Vendor Specific             Vendor Specific
        other    any   Reserved                    Reserved"
        ::= { dot11RSNStatsEntry 4 }

    dot11RSNStatsTKIPICVErrors OBJECT-TYPE
        SYNTAX       Counter32
        MAX-ACCESS   read-only
        STATUS       current
        DESCRIPTION
            "Counts the number of TKIP ICV errors encountered when
            decrypting packets for the station."
        ::= { dot11RSNStatsEntry 5 }

    dot11RSNStatsTKIPLocalMICFailures OBJECT-TYPE
        SYNTAX       Counter32
        MAX-ACCESS   read-only
        STATUS       current
        DESCRIPTION
            "Counts the number of Michael MIC failure encountered when
            checking the integrity of packets received from the station
            at this entity."
        ::= { dot11RSNStatsEntry 6 }

    dot11RSNStatsTKIPRemoteMICFailures OBJECT-TYPE
        SYNTAX       Counter32
        MAX-ACCESS   read-only
        STATUS       current
        DESCRIPTION
            "Counts the number of Michael MIC failures encountered by
            the station identified by dot11StatsSTAAddress and reported
            back to this entity. "
        ::= { dot11RSNStatsEntry 7 }

    dot11RSNStatsTKIPCounterMeasuresInvoked OBJECT-TYPE
        SYNTAX       Counter32
        MAX-ACCESS   read-only
        STATUS       current
        DESCRIPTION
            "Counts the number of times a MIC failure occurred two times
            within 60 seconds and counter-measures were invoked. This
            variables counts this for both local and remote. It counts
            every time countermeasures are invoked. "
        ::= { dot11RSNStatsEntry 8 }

    dot11RSNStatsWRAPFormatErrors OBJECT-TYPE
        SYNTAX       Counter32
        MAX-ACCESS   read-only
        STATUS       current
        DESCRIPTION
            "The number of MSDUs received with an invalid WRAP format."
        ::= { dot11RSNStatsEntry 9 }

    dot11RSNStatsWRAPReplays OBJECT-TYPE
        SYNTAX       Counter32
        MAX-ACCESS   read-only
        STATUS       current
        DESCRIPTION
            "The number of received unicast fragments discarded by the
            replay mechanism."
        ::= { dot11RSNStatsEntry 10 }

    dot11RSNStatsWRAPDecryptErrors OBJECT-TYPE
        SYNTAX       Counter32
        MAX-ACCESS   read-only
        STATUS       current
        DESCRIPTION
            "The number of received fragments discarded by the OCB
            decryption mechanism."
        ::= { dot11RSNStatsEntry 11 }
```

```
dot11RSNStatsCCMPFormatErrors OBJECT-TYPE
      SYNTAX       Counter32
      MAX-ACCESS   read-only
      STATUS       current
      DESCRIPTION
            "The number of MSDUs received with an invalid CCMP format."
      ::= { dot11RSNStatsEntry 12 }
dot11RSNStatsCCMPReplays OBJECT-TYPE
      SYNTAX       Counter32
      MAX-ACCESS   read-only
      STATUS       current
      DESCRIPTION
            "The number of received unicast fragments discarded by the
            replay mechanism."
      ::= { dot11RSNStatsEntry 13 }
dot11RSNStatsCCMPDecryptErrors OBJECT-TYPE
      SYNTAX       Counter32
      MAX-ACCESS   read-only
      STATUS       current
      DESCRIPTION
            "The number of received fragments discarded by the CCMP
            decryption algorithm."
      ::= { dot11RSNStatsEntry 14 }
--
--   TBD: OBJECT-GROUPs and MODULE-COMPLIANCE statements
--
END
```

Annex F (informative) RSN reference implementations and test vectors

F.1 TKIP Temporal Key Mixing Function reference implementation and test vector

This clause provides a “C” language reference implementation of the temporal key mixing function.

```

/*****************************************************************/
/* 802.11 TKIP Key Mixing Test Vector Generator                  */
/*                                                               */
/* The author has put this code in the public domain.            */
/*                                                               */
/* Author: David Johnston                                        */
/* Email: dj@mobilian.com                                        */
/* Version 0.1                                                   */
/*                                                               */
/*****************************************************************/

#include <stdlib.h>
#include <stdio.h>

/*******************************************/
/* Test Cases                              */
/* An array of test cases                  */
/*******************************************/
#define NUM_TEST_CASES  8


unsigned long int test_case_pnl[] = { /* 2 lsbs of pn */
      0x0000,
```

```
                0x0001,
                0xffff,
                0x0000,
                0x058c,
                0x058d,
                0x30f8,
                0x30f9
        };
        unsigned long int test_case_pnh[] = { /* 4 msbs of pn */
                0x00000000,
                0x00000000,
                0x20dcfd43,
                0x20dcfd44,
                0xf0a410fc,
                0xf0a410fc,
                0x8b1573b7,
                0x8b1573b7
        };

        unsigned char keys[] =
        {
                0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,
                0x08,0x09,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F,
                0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,
                0x08,0x09,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F,
                0x63,0x89,0x3B,0x25,0x08,0x40,0xB8,0xAE,
                0x0B,0xD0,0xFA,0x7E,0x61,0xD2,0x78,0x3E,
                0x63,0x89,0x3B,0x25,0x08,0x40,0xB8,0xAE,
                0x0B,0xD0,0xFA,0x7E,0x61,0xD2,0x78,0x3E,
                0x98,0x3A,0x16,0xEF,0x4F,0xAC,0xB3,0x51,
                0xAA,0x9E,0xCC,0x27,0x1D,0x73,0x09,0xE2,
                0x98,0x3A,0x16,0xEF,0x4F,0xAC,0xB3,0x51,
                0xAA,0x9E,0xCC,0x27,0x1D,0x73,0x09,0xE2,
                0xC8,0xAD,0xC1,0x6A,0x8B,0x4D,0xDA,0x3B,
                0x4D,0xD5,0xB6,0x54,0x38,0x35,0x9B,0x05,
                0xC8,0xAD,0xC1,0x6A,0x8B,0x4D,0xDA,0x3B,
                0x4D,0xD5,0xB6,0x54,0x38,0x35,0x9B,0x05
        };

        unsigned char transmitter_addr[] =
        {
                0x10,0x22,0x33,0x44,0x55,0x66,
                0x10,0x22,0x33,0x44,0x55,0x66,
                0x64,0xF2,0xEA,0xED,0xDC,0x25,
                0x64,0xF2,0xEA,0xED,0xDC,0x25,
                0x50,0x9C,0x4B,0x17,0x27,0xD9,
                0x50,0x9C,0x4B,0x17,0x27,0xD9,
                0x94,0x5E,0x24,0x4E,0x4D,0x6E,
                0x94,0x5E,0x24,0x4E,0x4D,0x6E
        };

        /* The Sbox is reduced to 2 16-bit wide tables, each with 256 entries. */
        /* The 2nd table is the same as the 1st but with the upper and lower   */
        /* bytes swapped. To allow an endian tolerant implementation, the byte */
        /* halves have been expressed independently here.                      */

        unsigned int Tkip_Sbox_Lower[256] =
        {
                0xA5,0x84,0x99,0x8D,0x0D,0xBD,0xB1,0x54,
                0x50,0x03,0xA9,0x7D,0x19,0x62,0xE6,0x9A,
                0x45,0x9D,0x40,0x87,0x15,0xEB,0xC9,0x0B,
                0xEC,0x67,0xFD,0xEA,0xBF,0xF7,0x96,0x5B,
                0xC2,0x1C,0xAE,0x6A,0x5A,0x41,0x02,0x4F,
                0x5C,0xF4,0x34,0x08,0x93,0x73,0x53,0x3F,
                0x0C,0x52,0x65,0x5E,0x28,0xA1,0x0F,0xB5,
                0x09,0x36,0x9B,0x3D,0x26,0x69,0xCD,0x9F,
                0x1B,0x9E,0x74,0x2E,0x2D,0xB2,0xEE,0xFB,
```

```
        0xF6,0x4D,0x61,0xCE,0x7B,0x3E,0x71,0x97,
        0xF5,0x68,0x00,0x2C,0x60,0x1F,0xC8,0xED,
        0xBE,0x46,0xD9,0x4B,0xDE,0xD4,0xE8,0x4A,
        0x6B,0x2A,0xE5,0x16,0xC5,0xD7,0x55,0x94,
        0xCF,0x10,0x06,0x81,0xF0,0x44,0xBA,0xE3,
        0xF3,0xFE,0xC0,0x8A,0xAD,0xBC,0x48,0x04,
        0xDF,0xC1,0x75,0x63,0x30,0x1A,0x0E,0x6D,
        0x4C,0x14,0x35,0x2F,0xE1,0xA2,0xCC,0x39,
        0x57,0xF2,0x82,0x47,0xAC,0xE7,0x2B,0x95,
        0xA0,0x98,0xD1,0x7F,0x66,0x7E,0xAB,0x83,
        0xCA,0x29,0xD3,0x3C,0x79,0xE2,0x1D,0x76,
        0x3B,0x56,0x4E,0x1E,0xDB,0x0A,0x6C,0xE4,
        0x5D,0x6E,0xEF,0xA6,0xA8,0xA4,0x37,0x8B,
        0x32,0x43,0x59,0xB7,0x8C,0x64,0xD2,0xE0,
        0xB4,0xFA,0x07,0x25,0xAF,0x8E,0xE9,0x18,
        0xD5,0x88,0x6F,0x72,0x24,0xF1,0xC7,0x51,
        0x23,0x7C,0x9C,0x21,0xDD,0xDC,0x86,0x85,
        0x90,0x42,0xC4,0xAA,0xD8,0x05,0x01,0x12,
        0xA3,0x5F,0xF9,0xD0,0x91,0x58,0x27,0xB9,
        0x38,0x13,0xB3,0x33,0xBB,0x70,0x89,0xA7,
        0xB6,0x22,0x92,0x20,0x49,0xFF,0x78,0x7A,
        0x8F,0xF8,0x80,0x17,0xDA,0x31,0xC6,0xB8,
        0xC3,0xB0,0x77,0x11,0xCB,0xFC,0xD6,0x3A
};

unsigned int Tkip_Sbox_Upper[256] =
{
        0xC6,0xF8,0xEE,0xF6,0xFF,0xD6,0xDE,0x91,
        0x60,0x02,0xCE,0x56,0xE7,0xB5,0x4D,0xEC,
        0x8F,0x1F,0x89,0xFA,0xEF,0xB2,0x8E,0xFB,
        0x41,0xB3,0x5F,0x45,0x23,0x53,0xE4,0x9B,
        0x75,0xE1,0x3D,0x4C,0x6C,0x7E,0xF5,0x83,
        0x68,0x51,0xD1,0xF9,0xE2,0xAB,0x62,0x2A,
        0x08,0x95,0x46,0x9D,0x30,0x37,0x0A,0x2F,
        0x0E,0x24,0x1B,0xDF,0xCD,0x4E,0x7F,0xEA,
        0x12,0x1D,0x58,0x34,0x36,0xDC,0xB4,0x5B,
        0xA4,0x76,0xB7,0x7D,0x52,0xDD,0x5E,0x13,
        0xA6,0xB9,0x00,0xC1,0x40,0xE3,0x79,0xB6,
        0xD4,0x8D,0x67,0x72,0x94,0x98,0xB0,0x85,
        0xBB,0xC5,0x4F,0xED,0x86,0x9A,0x66,0x11,
        0x8A,0xE9,0x04,0xFE,0xA0,0x78,0x25,0x4B,
        0xA2,0x5D,0x80,0x05,0x3F,0x21,0x70,0xF1,
        0x63,0x77,0xAF,0x42,0x20,0xE5,0xFD,0xBF,
        0x81,0x18,0x26,0xC3,0xBE,0x35,0x88,0x2E,
        0x93,0x55,0xFC,0x7A,0xC8,0xBA,0x32,0xE6,
        0xC0,0x19,0x9E,0xA3,0x44,0x54,0x3B,0x0B,
        0x8C,0xC7,0x6B,0x28,0xA7,0xBC,0x16,0xAD,
        0xDB,0x64,0x74,0x14,0x92,0x0C,0x48,0xB8,
        0x9F,0xBD,0x43,0xC4,0x39,0x31,0xD3,0xF2,
        0xD5,0x8B,0x6E,0xDA,0x01,0xB1,0x9C,0x49,
        0xD8,0xAC,0xF3,0xCF,0xCA,0xF4,0x47,0x10,
        0x6F,0xF0,0x4A,0x5C,0x38,0x57,0x73,0x97,
        0xCB,0xA1,0xE8,0x3E,0x96,0x61,0x0D,0x0F,
        0xE0,0x7C,0x71,0xCC,0x90,0x06,0xF7,0x1C,
        0xC2,0x6A,0xAE,0x69,0x17,0x99,0x3A,0x27,
        0xD9,0xEB,0x2B,0x22,0xD2,0xA9,0x07,0x33,
        0x2D,0x3C,0x15,0xC9,0x87,0xAA,0x50,0xA5,
        0x03,0x59,0x09,0x1A,0x65,0xD7,0x84,0xD0,
        0x82,0x29,0x5A,0x1E,0x7B,0xA8,0x6D,0x2C
};


/*****************************/
/**** Function Prototypes ****/
/*****************************/

unsigned int tkip_sbox(unsigned int index);
```

```
unsigned int rotr1(unsigned int a);

/* Mixes key from TA, TK and TSC */

void mix_key(
      unsigned char   *key,
      unsigned char   *ta,
      unsigned long int pnl,  /* Least significant 16 bits of PN */
      unsigned long int pnh,  /* Most significant 32 bits of PN */
      unsigned char   *rc4key,
      unsigned int *p1kout
);

/*************************************************************/
/* tkip_sbox()                                               */
/* Returns a 16 bit value from a 64K entry table. The Table */
/* is synthesized from two 256 entry byte wide tables.      */
/*************************************************************/

unsigned int tkip_sbox(unsigned int index)
{
      unsigned int index_low;
      unsigned int index_high;
      unsigned int left, right;

      index_low = (index % 256);
      index_high = ((index >> 8) % 256);

      left = Tkip_Sbox_Lower[index_low] + (Tkip_Sbox_Upper[index_low] *
            256);
      right = Tkip_Sbox_Upper[index_high] + (Tkip_Sbox_Lower[index_high]
            * 256);

      return (left ^ right);
};

/***************************************************/
/* mix_key()                                       */
/* Takes a key, PN and TK. Calculates an RC4 key.  */
/***************************************************/

unsigned int rotr1(unsigned int a)
{
      unsigned int b;

      if ((a & 0x01) == 0x01)
      {
            b = (a >> 1) | 0x8000;
      }
      else
      {
            b = (a >> 1) & 0x7fff;
      }
      b = b % 65536;
      return b;
}

void mix_key(
      unsigned char   *key,
      unsigned char   *ta,
      unsigned long int pnl,  /* Least significant 16 bits of PN */
      unsigned long int pnh,  /* Most significant 32 bits of PN */
      unsigned char   *rc4key,
      unsigned int *p1k
      )
{
      unsigned int ttak0;    /* 16 bit numbers */
```

```
        unsigned int ttak1;
        unsigned int ttak2;
        unsigned int ttak3;
        unsigned int ttak4;

        unsigned int tsc0;
        unsigned int tsc1;
        unsigned int tsc2;

        unsigned int ppk0;
        unsigned int ppk1;
        unsigned int ppk2;
        unsigned int ppk3;
        unsigned int ppk4;
        unsigned int ppk5;

        int i;
        int j;

        tsc0 = (unsigned int)((pnh >> 16) % 65536);  /* msb */
        tsc1 = (unsigned int)(pnh % 65536);
        tsc2 = (unsigned int)(pnl % 65536); /* lsb */

        /* Phase 1, step 1 */
        p1k[0] = tsc1;
        p1k[1] = tsc0;
        p1k[2] = (unsigned int)(ta[0] + (ta[1]*256));
        p1k[3] = (unsigned int)(ta[2] + (ta[3]*256));
        p1k[4] = (unsigned int)(ta[4] + (ta[5]*256));

        /* Phase 1, step 2 */
        for (i=0; i<8; i++)
        {
              j = 2*(i & 1);
              p1k[0] =  (p1k[0] + tkip_sbox( (p1k[4] ^ ((256*key[1+j]) +
              key[j])) % 65536 )) % 65536;
              p1k[1] =  (p1k[1] + tkip_sbox( (p1k[0] ^ ((256*key[5+j]) +
              key[4+j])) % 65536 )) % 65536;
              p1k[2] =  (p1k[2] + tkip_sbox( (p1k[1] ^ ((256*key[9+j]) +
              key[8+j])) % 65536 )) % 65536;
              p1k[3] =  (p1k[3] + tkip_sbox( (p1k[2] ^ ((256*key[13+j]) +
              key[12+j])) % 65536 )) % 65536;
              p1k[4] =  (p1k[4] + tkip_sbox( (p1k[3] ^ (((256*key[1+j]) +
              key[j]))) % 65536 )) % 65536;
              p1k[4] = (p1k[4] + i) % 65536;
        }

        /* Phase 2, Step 1 */
        ppk0 = p1k[0];
        ppk1 = p1k[1];
        ppk2 = p1k[2];
        ppk3 = p1k[3];
        ppk4 = p1k[4];
        ppk5 = (p1k[4] + tsc2) % 65536;

        /* Phase2, Step 2 */
        ppk0 = ppk0 + tkip_sbox( (ppk5 ^ ((256*key[1]) + key[0])) %
              65536);
        ppk1 = ppk1 + tkip_sbox( (ppk0 ^ ((256*key[3]) + key[2])) %
              65536);
        ppk2 = ppk2 + tkip_sbox( (ppk1 ^ ((256*key[5]) + key[4])) %
              65536);
        ppk3 = ppk3 + tkip_sbox( (ppk2 ^ ((256*key[7]) + key[6])) %
              65536);
        ppk4 = ppk4 + tkip_sbox( (ppk3 ^ ((256*key[9]) + key[8])) %
              65536);
```

```
        ppk5 = ppk5 + tkip_sbox( (ppk4 ^ ((256*key[11]) + key[10])) %
              65536);

        ppk0 = ppk0 + rotr1(ppk5 ^ ((256*key[13]) + key[12]));
        ppk1 = ppk1 + rotr1(ppk0 ^ ((256*key[15]) + key[14]));
        ppk2 = ppk2 + rotr1(ppk1);
        ppk3 = ppk3 + rotr1(ppk2);
        ppk4 = ppk4 + rotr1(ppk3);
        ppk5 = ppk5 + rotr1(ppk4);

        /* Phase 2, Step 3 */
        rc4key[0] = (tsc2 >> 8) % 256;
        rc4key[1] = (((tsc2 >> 8) % 256) | 0x20) & 0x7f;
        rc4key[2] = tsc2 % 256;
        rc4key[3] = ((ppk5 ^ ((256*key[1]) + key[0])) >> 1) % 256;

        rc4key[4] = ppk0 % 256;
        rc4key[5] = (ppk0 >> 8) % 256;

        rc4key[6] = ppk1 % 256;
        rc4key[7] = (ppk1 >> 8) % 256;

        rc4key[8] = ppk2 % 256;
        rc4key[9] = (ppk2 >> 8) % 256;

        rc4key[10] = ppk3 % 256;
        rc4key[11] = (ppk3 >> 8) % 256;

        rc4key[12] = ppk4 % 256;
        rc4key[13] = (ppk4 >> 8) % 256;

        rc4key[14] = ppk5 % 256;
        rc4key[15] = (ppk5 >> 8) % 256;
}

/****************************************************/
/* main()                                           */
/* Iterate through the test cases, passing them     */
/* through the TKIP algorithm to produce test       */
/* vectors and verify decryption against encryption */
/****************************************************/

int main()
{
        int length;
        int test_case;
        int header_length;
        int payload_length;
        int a4_exists;
        int qc_exists;
        unsigned char plaintext_mpdu[3000];
        unsigned char ciphertext_mpdu[3000];
        /*unsigned char decrypted_mpdu[3000];*/
        unsigned char *key;
        unsigned char *ta;
        unsigned char rc4key[16];
        unsigned int p1k[5];
        unsigned long int iv32;
        unsigned int iv16;

        unsigned int i;

        for (i=0; i<16;i++) rc4key[i] = 0x00;

        for (test_case = 1; test_case < (NUM_TEST_CASES+1); test_case++)
        {
              printf ("\nTest vector #%d:\n",test_case);
```

```
			key = keys + (16 * (test_case-1));
			ta = transmitter_addr + (6 * (test_case-1));

			mix_key(key,
				ta,
				test_case_pnl[test_case-1],
				test_case_pnh[test_case-1],
				rc4key,
				p1k
				);

			printf("TK    =");
			for (i=0; i<16; i++)
			{
				printf(" %02X", key[i]);
			}
			printf(" [LSB on left, MSB on right]\n");

			printf("TA    =");
			printf(" %02X", ta[0]);
			for (i=1; i<6; i++)
			{
				printf("-%02X", ta[i]);
			}
			printf("\n");

			printf("PN    = %08X%04X [transmitted as: ",
				test_case_pnh[test_case-1],
				test_case_pnl[test_case-1]);
			printf(" %02X %02X %02X DefKeyID",
				(test_case_pnl[test_case-1] % 256),
				(rc4key[1]),
				((test_case_pnl[test_case-1] >> 8) % 256)
				);
			printf(" %02X %02X %02X %02X]\n",
				(test_case_pnh[test_case-1] % 256),
				((test_case_pnh[test_case-1] >> 8) % 256),
				((test_case_pnh[test_case-1] >> 16) % 256),
				((test_case_pnh[test_case-1] >> 24) % 256)
				);

			printf("IV32  = %08X\n", test_case_pnh[test_case-1]);
			printf("IV16  = %04X\n", test_case_pnl[test_case-1]);


			printf("P1K   =");
			for (i=0; i<5; i++)
			{
				printf(" %04X", p1k[i]);
			}
			printf("\n");

			printf("RC4KEY=");
			for (i=0; i<16; i++)
			{
				printf(" %02X", rc4key[i]);
			}
			printf("\n");

		}

		return 0;
	}
```

F.1.2 Test Vectors

The following output is provided to test implementations of the temporal key hash algorithm. All input and output values are shown in hexadecimal.

```
Test vector #1:
TK    = 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F [LSB on left, MSB on right]
TA    = 10-22-33-44-55-66
PN    = 000000000000 [transmitted as:  00 20 00 DefKeyID 00 00 00 00]
IV32  = 00000000   [transmitted as 00 00 00 00]
IV16  = 0000
P1K   = 3DD2  016E  76F4  8697  B2E8
RC4KEY= 00 20 00 33 EA 8D 2F 60 CA 6D 13 74 23 4A 66 0B

Test vector #2:
TK    = 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F [LSB on left, MSB on right]
TA    = 10-22-33-44-55-66
PN    = 000000000001 [transmitted as:  01 20 00 DefKeyID 00 00 00 00]
IV32  = 00000000   [transmitted as 00 00 00 00]
IV16  = 0001
P1K   = 3DD2  016E  76F4  8697  B2E8
RC4KEY= 00 20 01 90 FF DC 31 43 89 A9 D9 D0 74 FD 20 AA

Test vector #3:
TK    = 63 89 3B 25 08 40 B8 AE 0B D0 FA 7E 61 D2 78 3E [LSB on left, MSB on right]
TA    = 64-F2-EA-ED-DC-25
PN    = 20DCFD43FFFF [transmitted as:  FF 7F FF DefKeyID 43 FD DC 20]
IV32  = 20DCFD43   [transmitted as 20 DC FD 43]
IV16  = FFFF
P1K   = 7C67  49D7  9724  B5E9  B4F1
RC4KEY= FF 7F FF 93 81 0F C6 E5 8F 5D D3 26 25 15 44 CE

Test vector #4:
TK    = 63 89 3B 25 08 40 B8 AE 0B D0 FA 7E 61 D2 78 3E [LSB on left, MSB on right]
TA    = 64-F2-EA-ED-DC-25
PN    = 20DCFD440000 [transmitted as:  00 20 00 DefKeyID 44 FD DC 20]
IV32  = 20DCFD44   [transmitted as 20 DC FD 44]
IV16  = 0000
P1K   = 5A5D  73A8  A859  2EC1  DC8B
RC4KEY= 00 20 00 49 8C A4 71 FC FB FA A1 6E 36 10 F0 05

Test vector #5:
TK    = 98 3A 16 EF 4F AC B3 51 AA 9E CC 27 1D 73 09 E2 [LSB on left, MSB on right]
TA    = 50-9C-4B-17-27-D9
PN    = F0A410FC058C [transmitted as:  8C 25 05 DefKeyID FC 10 A4 F0]
IV32  = F0A410FC   [transmitted as F0 A4 10 FC]
IV16  = 058C
P1K   = F2DF  EBB1  88D3  5923  A07C
RC4KEY= 05 25 8C F4 D8 51 52 F4 D9 AF 1A 64 F1 D0 70 21

Test vector #6:
TK    = 98 3A 16 EF 4F AC B3 51 AA 9E CC 27 1D 73 09 E2 [LSB on left, MSB on right]
TA    = 50-9C-4B-17-27-D9
PN    = F0A410FC058D [transmitted as:  8D 25 05 DefKeyID FC 10 A4 F0]
IV32  = F0A410FC   [transmitted as F0 A4 10 FC]
IV16  = 058D
P1K   = F2DF  EBB1  88D3  5923  A07C
RC4KEY= 05 25 8D 09 F8 15 43 B7 6A 59 6F C2 C6 73 8B 30

Test vector #7:
TK    = C8 AD C1 6A 8B 4D DA 3B 4D D5 B6 54 38 35 9B 05 [LSB on left, MSB on right]
TA    = 94-5E-24-4E-4D-6E
PN    = 8B1573B730F8 [transmitted as:  F8 30 30 DefKeyID B7 73 15 8B]
IV32  = 8B1573B7   [transmitted as 8B 15 73 B7]
IV16  = 30F8
P1K   = EFF1  3F38  A364  60A9  76F3
RC4KEY= 30 30 F8 65 0D A0 73 EA 61 4E A8 F4 74 EE 03 19
```

```
Test vector #8:
TK    = C8 AD C1 6A 8B 4D DA 3B 4D D5 B6 54 38 35 9B 05 [LSB on left, MSB on right]
TA    = 94-5E-24-4E-4D-6E
PN    = 8B1573B730F9 [transmitted as:  F9 30 30 DefKeyID B7 73 15 8B]
IV32  = 8B1573B7   [transmitted as 8B 15 73 B7]
IV16  = 30F9
P1K   = EFF1  3F38  A364  60A9  76F3
RC4KEY= 30 30 F9 31 55 CE 29 34 37 CC 76 71 27 16 AB 8F
```

F.2 Michael reference implementation and test vectors

F.2.1 Michael test vectors

To ensure correct implementation of Michael, here are some test vectors. These test vectors still have to be confirmed by an independent implementation.

F.2.1.1 Block function

Here are some test vectors for the block function.

Input	# times	output
(00000000, 00000000)	1	(00000000, 00000000)
(00000000, 00000001)	1	(c00015a8, c0000b95)
(00000001, 00000000)	1	(6b519593, 572b8b8a)
(01234567, 83659326)	1	(441492c2, 1d8427ed)
(00000001, 00000000)	1000	(9f04c4ad, 2ec6c2bf)

The first four rows give test vectors for a single application of the block function *b*. The last row gives a test vector for 1000 repeated applications of the block function. Together these should provide adequate test coverage.

F.2.1.2 Michael

Here are some test vectors for Michael.

Key	message	output
0000000000000000	""	82925c1ca1d130b8
82925c1ca1d130b8	"M”	434721ca40639b3f
434721ca40639b3f	"Mi “	E8f9becae97e5d29
e8f9becae97e5d29	"Mic"	90038fc6cf13c1db
90038fc6cf13c1db	"Mich"	d55e100510128986
d55e100510128986	"Michael"	0a942b124ecaa546

Note that each key is the result of the previous line, which makes it easy to construct a single test out of all of these test cases.

F.2.2 Example code

```
//
// Michael.h    Reference implementation for Michael
//              written by Niels Ferguson
```

```
//
// The author has put this code in the public domain.
//

//
// A Michael object implements the computation of the Michael MIC.
//
// Conceptually, the object stores the message to be authenticated.
// At construction the message is empty.
// The append() method appends bytes to the message.
// The getMic() method computes the MIC over the message and returns the
result.
// As a side-effect it also resets the stored message
// to the empty message so that the object can be re-used
// for another MIC computation.

class Michael
      {

public:
      // Constructor requires a pointer to 8 bytes of key
      Michael( Byte * key );

      // Destructor
      ~Michael();

      // Clear the internal message,
      // resets the object to the state just after construction.
      void clear();

      // Set the key to a new value
      void setKey( Byte * key );

      // Append bytes to the message to be MICed
      void append( Byte * src, int nBytes );

      // Get the MIC result. Destination should accept 8 bytes of
      result.
      // This also resets the message to empty.
      void getMIC( Byte * dst );

      // Run the test plan to verify proper operations
      static void runTestPlan();

private:
      // Copy constructor declared but not defined,
      //avoids compiler-generated version.
      Michael( const Michael & );
      // Assignment operator declared but not defined,
      //avoids compiler-generated version.
      void operator=( const Michael & );


      // A bunch of internal functions

      // Get UInt32 from 4 bytes LSByte first
      static UInt32 getUInt32( Byte * p );

      // Put UInt32 into 4 bytes LSByte first
      static void putUInt32( Byte * p, UInt32 val );

      // Add a single byte to the internal message
      void appendByte( Byte b );


      // Conversion of hex string to binary string
      static void hexToBin( char *src, Byte * dst );
```

```
        // More conversion of hex string to binary string
        static void hexToBin( char *src, int nChars, Byte * dst );

        // Helper function for hex conversion
        static Byte hexToBinNibble( char c );

        // Run a single test case
        static void runSingleTest( char * cKey, char * cMsg, char *
        cResult );


        UInt32  K0, K1;         // Key
        UInt32  L, R;           // Current state
        UInt32  M;              // Message accumulator (single word)
        int     nBytesInM;      // # bytes in M
        };


//
// Michael.cpp  Reference implementation for Michael
//              written by Niels Ferguson
//
// The author has put this code in the public domain.// All rights
reserved,
//

// Adapt these typedefs to your local platform
typedef unsigned long UInt32;
typedef unsigned char Byte;

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "Michael.h"

// Rotation functions on 32 bit values
#define ROL32( A, n ) \
        ( ((A) << (n)) | ( ((A)>>(32-(n)))  & ( (1UL << (n)) - 1 ) ) )
#define ROR32( A, n ) ROL32( (A), 32-(n) )


UInt32 Michael::getUInt32( Byte * p )
        // Convert from Byte[] to UInt32 in a portable way
        {
        UInt32 res = 0;
        for( int i=0; i<4; i++ )
                {
                res |= (*p++) << (8*i);
                }
        return res;
}


void Michael::putUInt32( Byte * p, UInt32 val )
        // Convert from UInt32 to Byte[] in a portable way
        {
        for( int i=0; i<4; i++ )
                {
                *p++ = (Byte) (val & 0xff);
                val >>= 8;
                }
        }
```

```
void Michael::clear()
      {
      // Reset the state to the empty message.
      L = K0;
      R = K1;
      nBytesInM = 0;
      M = 0;
      }


void Michael::setKey( Byte * key )
      {
      // Set the key
      K0 = getUInt32( key );
      K1 = getUInt32( key + 4 );
      // and reset the message
      clear();
      }


Michael::Michael( Byte * key )
      {
      setKey( key );
      }


Michael::~Michael()
      {
      // Wipe the key material
      K0 = 0;
      K1 = 0;

      // And the other fields as well.
      //Note that this sets (L,R) to (K0,K1) which is just fine.
      clear();
      }


void Michael::appendByte( Byte b )
      {
      // Append the byte to our word-sized buffer
      M |= b << (8*nBytesInM);
      nBytesInM++;
      // Process the word if it is full.
      if( nBytesInM >= 4 )
            {
            L ^= M;
            R ^= ROL32( L, 17 );
            L += R;
            R ^= ((L & 0xff00ff00) >> 8) | ((L & 0x00ff00ff) << 8);
            L += R;
            R ^= ROL32( L, 3 );
            L += R;
            R ^= ROR32( L, 2 );
            L += R;
            // Clear the buffer
            M = 0;
            nBytesInM = 0;
            }
      }


void Michael::append( Byte * src, int nBytes )
      {
      // This is simple
      while( nBytes > 0 )
            {
```

```
                appendByte( *src++ );
                nBytes--;
                }
        }


void Michael::getMIC( Byte * dst )
        {
        // Append the minimum padding
        appendByte( 0x5a );
        appendByte( 0 );
        appendByte( 0 );
        appendByte( 0 );
        appendByte( 0 );
        // and then zeroes until the length is a multiple of 4
        while( nBytesInM != 0 )
                {
                appendByte( 0 );
                }
        // The appendByte function has already computed the result.
        putUInt32( dst, L );
        putUInt32( dst+4, R );
        // Reset to the empty message.
        clear();
        }


void Michael::hexToBin( char *src, Byte * dst )
        {
        // Simple wrapper
        hexToBin( src, strlen( src ), dst );
        }


void Michael::hexToBin( char *src, int nChars, Byte * dst )
        {
        assert( (nChars & 1) == 0 );
        int nBytes = nChars/2;

        // Straightforward conversion
        for( int i=0; i<nBytes; i++ )
                {
                dst[i] = (Byte)((hexToBinNibble( src[0] ) << 4)
                        | hexToBinNibble( src[1] ));
                src += 2;
                }
        }


Byte Michael::hexToBinNibble( char c )
        {
        if( '0' <= c && c <= '9' )
                {
                return (Byte)(c - '0');
                }
        // Make it upper case
        c &= ~('a'-'A');

        assert( 'A' <= c && c <= 'F' );
        return (Byte)(c - 'A' + 10);
        }


void Michael::runSingleTest( char * cKey, char * cMsg, char * cResult )
        {
        Byte key[ 8 ];
        Byte result[ 8 ];
```

```
		Byte res[ 8 ];

		// Convert key and result to binary form
		hexToBin( cKey, key );
		hexToBin( cResult, result );

		// Compute the MIC value
		Michael mic( key );
		mic.append( (Byte *)cMsg, strlen( cMsg) );
		mic.getMIC( res );

		// Check that it matches
		assert( memcmp( res, result, 8 ) == 0 );
		}


	void Michael::runTestPlan()
		// As usual, test plans can be quite tedious but this should
		// ensure that the implementation runs as expected.
		{
		Byte key[8] ;
		Byte msg[12];
		int i;

		// First we test the test vectors for the block function

		// The case (0,0)
		putUInt32( key, 0 );
		putUInt32( key+4, 0 );
		putUInt32( msg, 0 );

		Michael mic( key );
		mic.append( msg, 4 );

		assert( mic.L == 0 && mic.R == 0 );

		// The case (0,1)
		putUInt32( key, 0 );
		putUInt32( key+4, 1 );
		mic.setKey( key );
		mic.append( msg, 4 );

		assert( mic.L == 0xc00015a8 && mic.R == 0xc0000b95 );

		// The case (1,0)
		putUInt32( key, 1 );
		putUInt32( key+4, 0 );
		mic.setKey( key );
		mic.append( msg, 4 );

		assert( mic.L == 0x6b519593 && mic.R == 0x572b8b8a );

		// The case (01234567, 83659326)
		putUInt32( key, 0x01234567 );
		putUInt32( key+4, 0x83659326 );
		mic.setKey( key );
		mic.append( msg, 4 );

		assert( mic.L == 0x441492c2 && mic.R == 0x1d8427ed );

		// The repeated case
		putUInt32( key, 1 );
		putUInt32( key+4,0 );
		mic.setKey( key );

		for( i=0; i<1000; i++ )
			{
```

```
                mic.append( msg, 4 );
                }

        assert( mic.L == 0x9f04c4ad && mic.R == 0x2ec6c2bf );

        // And now for the real test cases
        runSingleTest( "0000000000000000", ""        , "82925c1ca1d130b8" );
        runSingleTest( "82925c1ca1d130b8", "M"       , "434721ca40639b3f" );
        runSingleTest( "434721ca40639b3f", "Mi"      , "e8f9becae97e5d29" );
        runSingleTest( "e8f9becae97e5d29", "Mic"     , "90038fc6cf13c1db" );
        runSingleTest( "90038fc6cf13c1db", "Mich"    , "d55e100510128986" );
        runSingleTest( "d55e100510128986", "Michael" , "0a942b124ecaa546" );
        }
```

F.3 HMAC-MD5 reference implementation and test vectors

F.3.1 Reference code

```
#include "stdafx.h"
#define ULONG unsigned long
#include <md5.h>

/*
 * Function: hmac_md5 from rfc2104; uses an MD5 library
 */

void hmac_md5(
        unsigned char *text, int text_len,
        unsigned char *key, int key_len,
        void * digest)
{
        MD5_CTX context;
        unsigned char k_ipad[65]; /* inner padding - key XORd with ipad */
        unsigned char k_opad[65]; /* outer padding - key XORd with opad */
        int i;

        /* if key is longer than 64 bytes reset it to key=MD5(key) */
        if (key_len > 64) {
                MD5_CTX tctx;

                MD5Init(&tctx);
                MD5Update(&tctx, key, key_len);
                MD5Final(&tctx);

                key = tctx.digest;
                key_len = 16;
        }

        /*
         * the HMAC_MD5 transform looks like:
         *
         *   MD5(K XOR opad, MD5(K XOR ipad, text))
         *
         * where K is an n byte key
         * ipad is the byte 0x36 repeated 64 times
         * opad is the byte 0x5c repeated 64 times
```

```
         * and text is the data being protected
         */

        /* start out by storing key in pads */
        memset(k_ipad, 0, sizeof k_ipad);
        memset(k_opad, 0, sizeof k_opad);
        memcpy(k_ipad, key, key_len);
        memcpy(k_opad, key, key_len);

        /* XOR key with ipad and opad values */
        for (i = 0; i < 64; i++) {
               k_ipad[i] ^= 0x36;
               k_opad[i] ^= 0x5c;
        }

        /* perform inner MD5 */
        MD5Init(&context); /* init context for 1st pass */
        MD5Update(&context, k_ipad, 64);  /* start with inner pad*/
        MD5Update(&context, text, text_len); /* then text of datagram */
        MD5Final(&context); /* finish up 1st pass */
        memcpy(digest, context.digest, 16);

        /* perform outer MD5 */
        MD5Init(&context); /* init context for 2nd pass */
        MD5Update(&context, (const unsigned char*)k_opad, 64);
               /* start with outer pad */
        MD5Update(&context, (const unsigned char*)digest, 16);
               /* then results of 1st hash */
        MD5Final(&context); /* finish up 2nd pass */
        memcpy(digest, context.digest, 16);
}
```

F.3.2 Test vectors

Test case 1	
Key	0x0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b
Key length	16
Data	"Hi There"
data_length	8
digest	0x9294727a3638bb1c13f48ef8158bfc9d

Test case 2	
Key	"Jefe"
Key length	4
Data	"what do ya want for nothing?"
Data length	28
Digest	0x750c783e6ab0b503eaa86e310a5db738

Test case 3	
Key	0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Key length	16
Data	0xdd repeated 50 times
Data length	50
Digest	0x56be34521d144c88dbb8c733f0e8b3f6

Test case 4	
Key	0x0102030405060708090a0b0c0d0e0f10111213141516171819
Key length	25
Data	0xcd repeated 50 times
Data length	50
Digest	0x697eaf0aca3a3aea3a75164746ffaa79

Test case 5
Key 0x0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c
Key length 16
Data "Test With Truncation"
Data length 20
Digest 0x56461ef2342edc00f9bab995690efd4c
Digest-96 0x56461ef2342edc00f9bab995

Test case 6
Key 0xaa repeated 80 times
Key length 80
Data "Test Using Larger Than Block-Size Key - Hash Key First"
Data length 54
Digest 0x6b1ab7fe4bd7bf8f0b62e6ce61b9d0cd

Test case 7
Key 0xaa repeated 80 times
Key length 80
Data "Test Using Larger Than Block-Size Key and Larger Than One Block-Size Data"
Data length 73
Digest 0x6f630fad67cda0ee1fb1f562db3aa53e

F.4 HMAC-SHA1 reference implementation and test vectors

F.4.1 HMAC-SHA1 Reference code

```
#include "stdafx.h"
#define ULONG unsigned long
#include <sha.h>

void hmac_sha1(
        unsigned char *text, int text_len,
        unsigned char *key, int key_len,
        unsigned char *digest)
{
        A_SHA_CTX context;
        unsigned char k_ipad[65]; /* inner padding - key XORd with ipad */
        unsigned char k_opad[65]; /* outer padding - key XORd with opad */
        int i;

        /* if key is longer than 64 bytes reset it to key=SHA1(key) */
        if (key_len > 64) {
                A_SHA_CTX      tctx;

                A_SHAInit(&tctx);
                A_SHAUpdate(&tctx, key, key_len);
                A_SHAFinal(&tctx, key);

                key_len = 20;
        }

        /*
         * the HMAC_SHA1 transform looks like:
         *
         * SHA1(K XOR opad, SHA1(K XOR ipad, text))
         *
         * where K is an n byte key
         * ipad is the byte 0x36 repeated 64 times
         * opad is the byte 0x5c repeated 64 times
```

```
                 * and text is the data being protected
                 */

                /* start out by storing key in pads */
                memset(k_ipad, 0, sizeof k_ipad);
                memset(k_opad, 0, sizeof k_opad);
                memcpy(k_ipad, key, key_len);
                memcpy(k_opad, key, key_len);

                /* XOR key with ipad and opad values */
                for (i = 0; i < 64; i++) {
                        k_ipad[i] ^= 0x36;
                        k_opad[i] ^= 0x5c;
                }

                /* perform inner SHA1*/
                A_SHAInit(&context); /* init context for 1st pass */
                A_SHAUpdate(&context, k_ipad, 64); /* start with inner pad */
                A_SHAUpdate(&context, text, text_len); /* then text of datagram */
                A_SHAFinal(&context, digest); /* finish up 1st pass */

                /* perform outer SHA1 */
                A_SHAInit(&context); /* init context for 2nd pass */
                A_SHAUpdate(&context, k_opad, 64); /* start with outer pad */
                A_SHAUpdate(&context, digest, 20); /* then results of 1st hash */
                A_SHAFinal(&context, digest); /* finish up 2nd pass */
        }
```

F.4.2 HMAC-SHA1 Test vectors

Test case 1	
Key	0x0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b
Key length	20
Data	"Hi There"
Data length	8
Digest	0xb617318655057264e28bc0b6fb378c8ef146be00

Test case 2	
Key	"Jefe"
Key length	4
Data	"what do ya want for nothing?"
Data length	28
Digest	0xeffcdf6ae5eb2fa2d27416d5f184df9c259a7c79

Test case 3	
Key	0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Key length	20
Data	0xdd repeated 50 times
Data length	50
Digest	0x125d7342b9ac11cd91a39af48aa17b4f63f175d3

Test case 4	
Key	0x0102030405060708090a0b0c0d0e0f10111213141516171819
Key length	25
Data	0xcd repeated 50 times
Data length	50
Digest	0x4c9007f4026250c6bc8414f9bf50c86c2d7235dane 7

Test case 5	
Key	0x0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c
Key len	20

Data	"Test With Truncation"
Data len	20
Digest	0x4c1a03424b55e07fe7f27be1d58bb9324a9a5a04
Digest-96	0x4c1a03424b55e07fe7f27be1

Test case 6	
Key	0xaa repeated 80 times
Key length	80
Data	"Test Using Larger Than Block-Size Key - Hash Key First"
Data length	54
Digest	0xaa4ae5e15272d00e95705637ce8a3b55ed402112

Test case 7	
Key	0xaa repeated 80 times
Key length	80
Data	"Test Using Larger Than Block-Size Key and Larger Than One Block-Size Data"
Data length	73
digest = 0xe8e99d0f45237d786d6bbaa7965c7808bbff1a91	
Data length	20
Digest	0x4c1a03424b55e07fe7f27be1d58bb9324a9a5a04
Digest-96	0x4c1a03424b55e07fe7f27be1

Test case 6	
Key	0xaa repeated 80 times
Key length	80
Data	"Test Using Larger Than Block-Size Key - Hash Key First"
Data length	54
Digest	0xaa4ae5e15272d00e95705637ce8a3b55ed402112

Test case 7	
Key	0xaa repeated 80 times
Key length	80
Data	"Test Using Larger Than Block-Size Key and Larger Than One Block-Size Data"
Data length	73
Digest	0xe8e99d0f45237d786d6bbaa7965c7808bbff1a91

F.5 PRF reference implementation and test vectors

F.5.1 PRF Reference code

```
/*
 * PRF -- Length of output is in octets rather than bits
 *     since length is always a multiple of 8 output array is
 *     organized so first N octets starting from 0 contains PRF output
 *
 *     supported inputs are 16, 32, 48, 64
 *     output array must be 80 octets to allow for sha1 overflow
 */
void PRF(
        unsigned char *key, int key_len,
        unsigned char *prefix, int prefix_len,
        unsigned char *data, int data_len,
        unsigned char *output, int len)
{
        int i;
        unsigned char input[1024]; /* concatenated input */
        int currentindex = 0;
```

```
            int total_len;

            memcpy(input, prefix, prefix_len);
            input[prefix_len] = 0; /* single octet 0 */
            memcpy(&input[prefix_len+1], data, data_len);
            total_len = prefix_len + 1 + data_len;
            input[total_len] = 0; /* single octet count, starts at 0 */
            total_len++;
            for(i = 0; i < (len+19)/20; i++) {
                  hmac_sha1(input, total_len, key, key_len,
                        &output[currentindex]);
                  currentindex += 20; /* next concatenation location */
                  input[total_len-1]++; /* increment octet count */
            }
      }
```

F.5.2 PRF Test vectors

Test case 1	
Key	0x0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b
Key length	20
Prefix	“prefix”
Prefix length	6
Data	"Hi There"
Data length	8
PRF-512	
	0xbcd4c650b30b9684951829e0d75f9d54b862175ed9f00606e17d8da35402ffee75df78c3
	d31e0f889f012120c0862beb67753e7439ae242edb8373698356cf5a

Test case 2	
Key	"Jefe"
Key length	4
Prefix	“prefix”
Prefix length	6
Data	"what do ya want for nothing?"
Data length	28
PRF-512	
	0x51f4de5b33f249adf81aeb713a3c20f4fe631446fabdfa58244759ae58ef9009a99abf4eac
	2ca5fa87e692c440eb40023e7babb206d61de7b92f41529092b8fc

Test case 3	
Key	0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Key length	20
Prefix	“prefix”
Prefix length	6
Data	0xdd repeated 50 times
Data length	50
PRF-512	
	0xe1ac546ec4cb636f9976487be5c86be17a0252ca5d8d8df12cfb0473525249ce9dd8d177
	ead710bc9b590547239107aef7b4abd43d87f0a68f1cbd9e2b6f7607

F.6. OCB Mode

The contents of this clause have been reproduced by permission of Phil Rogaway.

F.6.1 OCB Definition

F.6.1.1 Notation

NOTATION. If a and b are integers, $a \le b$, then $[a..b]$ is the set $\{a, a+1, \ldots, b\}$. If $i \ge 1$ is an integer then $\mathsf{ntz}(i)$ is the number of trailing 0-bits in the binary representation of i (equivalently, $\mathsf{ntz}(i)$ is the largest integer z such that 2^z divides i. So, for example, $\mathsf{ntz}(7)=0$ and $\mathsf{ntz}(8)=3$.

A *string* is a finite sequence of symbols, each symbol being 0 or 1. The string of length 0 is called the *empty string* and is denoted ε. Let $\{0, 1\}^*$ denote the set of all strings. If $A, B \in \{0, 1\}^*$ then $A\ B$, or $A \| B$, is their concatenation. If $A \in \{0, 1\}^*$ and $A \ne \varepsilon$ then $\mathsf{firstbit}(A)$ is the first bit of A and $\mathsf{lastbit}(A)$ is the last bit of A. Let i, n be nonnegative integers. Then 0^i and 1^i denote the strings of i 0's and 1's, respectively. Let $\{0, 1\}^n$ denote the set of all strings of length n. If $A \in \{0, 1\}^*$ then $|A|$ denotes the length of A, in bits, while $\|A\|_n = \max\{1, \lceil |A|/n \rceil\}$ denotes the length of A in n-bit blocks, where the empty string counts as one block. For $A \in \{0, 1\}^*$ and $|A| \le n$, $\mathsf{padd}_n(A)$ is the string $A\ 0^{n-|A|}$. With n understood we will write $\mathsf{pad}\{A\}$ for $\mathsf{padd}_n(A)$. If $A \in \{0, 1\}^*$ and $\tau \in [0..|A|]$ then A[first τ bits] and A[last τ bits] denote the first τ bits of A and the last τ bits of A, respectively. Both of these values are the empty string if $\tau = 0$. If $A, B \in \{0, 1\}^*$ then $A \oplus B$ is the bit-wise xor of A[first l bits] and B[first l bits], where $l = \min\{|A|, |B|\}$ (where $\varepsilon \oplus A = A \oplus \varepsilon = \varepsilon$). So, for example, $1001 \oplus 11 = 01$. If $A = a_{n-1} \ldots a_1\ a_0 \in \{0, 1\}^n$ then $\mathsf{str2num}(A)$ is the number $2^{n-1} \cdot a_{n-1} + \ldots + 2^1 \cdot a_1 + 2^0 \cdot a_0$. If $a \in [0..2^n - 1]$ then $\mathsf{num2str}_n(a)$ is the n-bit string A such that $\mathsf{str2num}(A) = a$. Let $\mathsf{len}_n(A) = \mathsf{num2str}_n(|A|)$. We omit the subscript when n is understood.

If $A = a_{n-1}\ a_{n-2} \ldots a_1\ a_0 \in \{0, 1\}^n$ then $A \ll 1$ is the n-bit string $a_{n-2} \ldots a_1\ a_0\ 0$ which is a left shift of A by one bit (the first bit of A disappearing and a zero coming into the last bit), while $A \gg 1$ is the n-bit string $0\ a_{n-1}\ a_{n-2} \ldots a_1\ a_0$ which is a right shift of A by one bit (the last bit disappearing and a zero coming into the first bit).

In pseudo code we write "Partition M into $M[1] \ldots M[m]$"' as shorthand for "Let $m = \mathsf{len}(M)$ and let $M[1], \ldots, M[m]$ be strings such that $M[1] \ldots M[m] = M$ and $|M[i]| = n$ for $1 \le i < m$." We write "Partition C into $C[1] \ldots C[m]\ T$" as shorthand for "if $|\ C\ | < \tau$ then return INVALID. Otherwise, let $C = C$[first $|C| - \tau$ bits], $T = C$[last $|C| - \tau$ bits], let $m = \|C\|_n$, and let $C[1], \ldots, C[m]$ be strings such that $C[1] \ldots C[m] = C$ and $|C[i]| = n$ for $1 \le i < m$." Recall that $\|M\|_n = \max\{1, \lceil |M|/n \rceil\}$, so the empty string partitions into $m = 1$ block, that one block being the empty string.

THE FIELD WITH 2^n POINTS. Let $\mathsf{GF}(2^n)$ denote the field with 2^n points. We interchangeably think of a point a in $\mathsf{GF}(2^n)$ in any of the following ways: (1) as an abstract point in a field; (2) as an n-bit string $a_{n-1} \ldots a_1\ a_0 \in \{0, 1\}^n$; (3) as a formal polynomial $a(x) = a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$ with binary coefficients; (4) as an integer between 0 and 2^{n-1}, where the string $a \in \{0, 1\}^n$ corresponds to the number $\mathsf{str2num}(a)$. For example, one can regard the string $a = 0^{125}\,101$ as a 128-bit string, as the number 5, as the polynomial x^2+1, or as an abstract point in $\mathsf{GF}(2^{128})$. We write $a(x)$ instead of a if we wish to emphasize that we are thinking of a as a polynomial.

To add two points in $\mathsf{GF}(2^n)$, take their bit-wise xor. We denote this operation by $a \oplus b$. To multiply two points in the field, first fix an irreducible polynomial $p_n(x)$ having binary coefficients and degree n: say the lexicographically first polynomial among the irreducible degree n polynomials having a minimum number of nonzero coefficients. For $n = 128$, the indicated polynomial is $p_{128}(x) = x^{128} + x^7 + x^2 + x + 1$. A few other $p_n(x)$-values are $x^{64} + x^4 + x^3 + x + 1$ and $x^{96} + x^{10} + x^9 + x^6 + 1$ and $x^{160} + x^5 + x^3 + x^2 + 1$ and $x^{192} + x^7 + x^2 + x + 1$ and $x^{224} + x^9 + x^8 + x^3 + 1$ and $x^{256} + x^{10} + x^5 + x^2 + 1$. To multiply $a, b \in \mathsf{GF}(2^n)$, which we denote $a \cdot b$, regard a and b as polynomials $a(x) = a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$ and $b(x) = b_{n-1}x^{n-1} + \ldots + b_1 x + b_0$, form their product $c(x)$ over $\mathsf{GF}(2)$, and take the remainder one gets when dividing $c(x)$ by $p_n(x)$.

It is computationally simple to multiply $a \in \{0, 1\}^n$ by x. We illustrate the method for $n = 128$, in which case multiplying $a = a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$ by x yields $a_{n-1}x^n + \ldots + a_1 x^2 + a_0 x$. Thus, if the first bit of a is

0, then $a \cdot x = a \ll 1$. If the first bit of a is 1 then we must add x^{128} to $a \ll 1$. Since $p_{128}(x) = x^{128} + x^7 + x^2 + x + 1 = 0$ we know that $x^{128} = x^7 + x^2 + x + 1$, so adding x^{128} means to xor by $0^{120}10000111$. In summary, when $n = 128$,

$$a \cdot x = \begin{cases} a \ll 1 & \text{if firstbit}(a) = 0 \\ (a \ll 1) \oplus 0^{120}10000111 & \text{if firstbit}(a) = 1 \end{cases}$$

It is similarly easy to divide $a \in \{0, 1\}^{128}$ by x (i.e., to multiply a by the multiplicative inverse of x). If the last bit of a is 0, then $a \cdot x^{-1}$ is $a \gg 1$. If the last bit of a is 1 then we must add (xor) to $a \gg 1$ the value x^{-1}. Since $x^{128} = x^7 + x^2 + x + 1$ we have that $x^{-1} = x^{127} + x^6 + x + 1 = 10^{120}\,1000011$. In summary, when $n = 128$,

$$a \cdot x^{-1} = \begin{cases} a \ll 1 & \text{if lastbit}(a) = 0 \\ (a \ll 1) \oplus 10^{120}1000011 & \text{if lastbit}(a) = 1 \end{cases}$$

If $L \in \{0, 1\}^n$ and $i \ge -1$, we write $L(i)$ as shorthand for $L \cdot x^i$. Using the equations just given, we have an easy way to compute from L the values $L(-1), L(0), L(1), \ldots, L(\mu)$, where μ is small number.

GRAY CODES. For $l \ge 1$, a Gray code is an ordering $\gamma^l = (\gamma^l_0, \gamma^l_1, \ldots, \gamma^l_k)$ of $\{0, 1\}^l$, where $k = 2^l-1$, such that successive points differ (in the Hamming sense) by just one bit. For n a fixed number, OCB makes use of the "canonical" Gray code $\gamma = \gamma^n$ constructed by $\gamma^1 = (0\ 1)$ and, for $l > 0$,

$$\gamma^{l+1} = (0\gamma^l_0\ \ 0\gamma l_1 \ldots 0\gamma^l_k\ \ 1\gamma^l_0\ \ 1\gamma^l_1 \ldots 1\gamma^l_k),\ \ k = 2^l-2$$

It is easy to see that γ is a Gray code. What is more, for $1 \le i \le 2^n-1$, $\gamma_i = \gamma_{i-1} \oplus (0^{n-1}1 \ll \mathrm{ntz}(i))$. This makes it easy to compute successive points.

We emphasize the following characteristics of the Gray-code values $\gamma_0, \gamma_1, \ldots, \gamma_k$, where $k = 2^n-1$: that they are distinct and different from 0; that $\gamma_1=1$; and that $\gamma_i < 2i$.

Let $L \in \{0, 1\}^n$ and consider the problem of successively forming the strings $\gamma_1 \cdot L, \gamma_2 \cdot L, \gamma_3 \cdot L, \ldots, \gamma_m \cdot L$. Of course $\gamma_1 \cdot L = 1 \cdot L = L$. Now, for $i \ge 2$, assume one has already produced $\gamma_{i-1} \cdot L$. Since $\gamma_i = \gamma_{i-1} \oplus (0^{n-1}1 \ll \mathrm{ntz}(i))$ we know that

$$\begin{aligned} \gamma_i \cdot L &= (\gamma_{i-1} \oplus (0^{n-1}1 \ll \mathrm{ntz}(i))) \cdot L \\ &= (\gamma_{i-1} \cdot L) \oplus (0^{n-1}1 \ll \mathrm{ntz}(i)) \cdot L \\ &= (\gamma_{i-1} \cdot L) \oplus (L \cdot x^{\mathrm{ntz}(i)}) \\ &= (\gamma_{i-1} \cdot L) \oplus L(\mathrm{ntz}(i)) \end{aligned}$$

That is, the ith word in the sequence $\gamma_1 \cdot L, \gamma_2 \cdot L, \gamma_3 \cdot L, \ldots$ is obtained by xoring the previous word with $L(\mathrm{ntz}(i))$. Had the sequence we were considering been $\gamma_1 \cdot L \oplus R, \gamma_2 \cdot L \oplus R, \gamma_3 \cdot L \oplus R, \ldots$, the ith word would be formed in the same way for $i \ge 2$, but the first word in the sequence would have been $L \oplus R$ instead of L.

F.6.1.2 The Scheme

PARAMETERS. To use OCB one must specify a block cipher and a tag length. The *block cipher* is a function $E: K \times \{0, 1\}^n \to \{0, 1\}^n$, for some number n, where each $E(K,\cdot) = E_K(\cdot)$ is a permutation on {0,

$1\}^n$. Here K is the set of possible keys and n is the block length. Both are arbitrary, though we insist that $n \geq 64$, and we discourage $n < 128$. The *tag length* is an integer $\tau \in [0..n]$. By trivial means, the adversary will be able to forge a valid ciphertext with probability $2^{-\tau}$. The popular block cipher to use with OCB is likely to be AES [34]. As for the tag length, a suggested default of $\tau = 64$ is reasonable. Tags of 32 bits are standard in retail banking. Tags of 96 bits are used in IPsec. Using a tag of more than 80 bits adds questionable security benefit, though it does lengthen each ciphertext.

We let OCB-E denote the OCB mode of operation using block cipher E and an unspecified tag length. We let OCB[E, τ] denote the OCB mode of operation using block cipher E and tag length τ.

NONCES. Encryption under OCB mode requires an n-bit nonce, N. The nonce would typically be a counter (maintained by the sender) or a random value (selected by the sender). Security is maintained even if the adversary can control the nonce, subject to the constraint that no nonce may be repeated within the current session (that is, during the period of use of the current encryption key). The nonce need not be random, unpredictable, or secret.

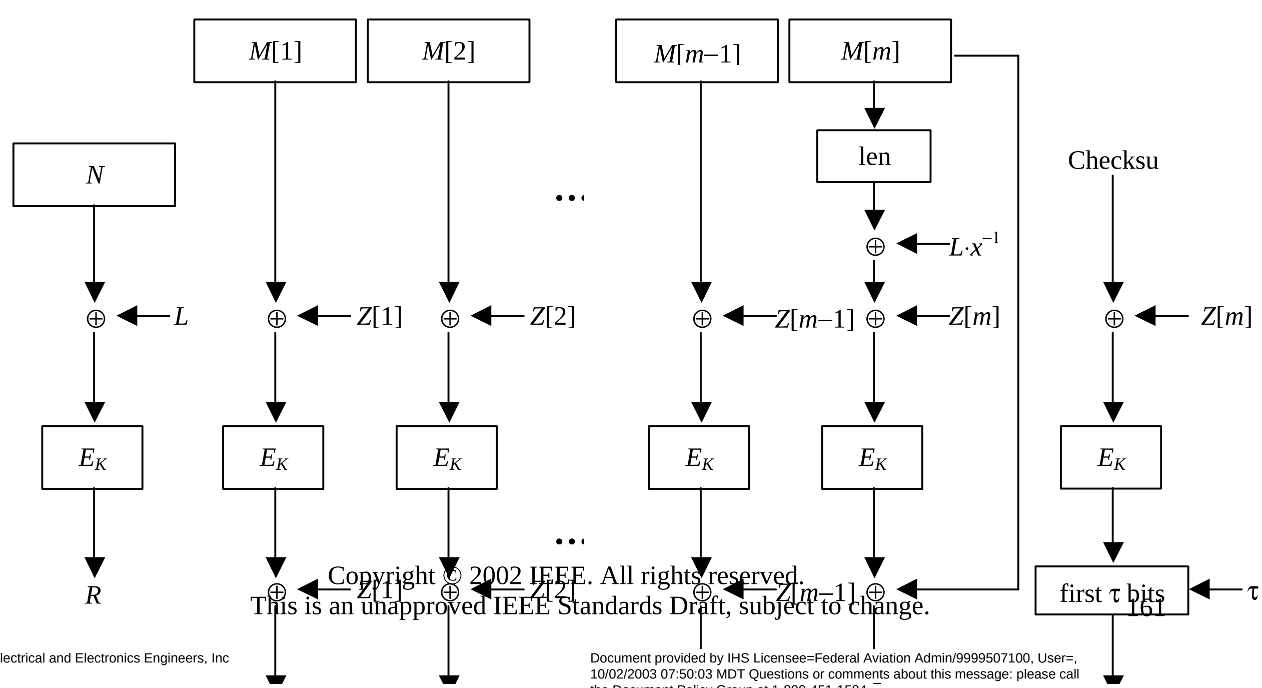
The nonce N is needed both to encrypt and to decrypt. Typically it would be communicated, in the clear, along with the ciphertext. However, it is out-of-scope how the nonce is communicated to the party who will decrypt. In particular, we do not regard the nonce as part of the ciphertext.

DEFINITION OF THE MODE. See Figure 54 for a definition and illustration of OCB. The figure defines OCB encryption and decryption. The key space for OCB is the key space K for the underlying block cipher E.

AN EQUIVALENT DESCRIPTION. The following description may clarify what a typical implementation might do.

Key generation. Choose a random key $K \leftarrow_R K$ for the block cipher. The key K is provided to both the entity that encrypts and the entity that decrypts.

Key setup. For the party that encrypts, do any key setup associated to block-cipher enciphering. For the party that decrypts, do any key setup associated to block-cipher enciphering and deciphering. Let $L \leftarrow E_K(0^n)$. Let m bound the maximum number of n-bit blocks that any message which will be encrypted or decrypted may have. Let $\mu \leftarrow \lceil \log_2 m \rceil$. Let $L(0) \leftarrow L$ and, for $i \in [1..\mu]$, compute $L(i) \leftarrow L(i-1) \cdot x$ using a shift and a conditional xor, as described in Section G.2. Compute $L(-1) \leftarrow L \cdot x^{-1}$ using a shift and a conditional xor, as described in Section G.2. Save the values $L(-1)$, $L(0)$, $L(1)$, …, $L(\mu)$ in a table.



Algorithm *OCB.Enc*$_{\mathbf{K}}$ *(*N, M*)*

Partition M into $M[1] \ldots M[m]$

$L \leftarrow E_K(0^n)$
$R \leftarrow E_K(N \oplus \mathrm{L})$
for $i \leftarrow 1$ **to** m **do** $Z[i] \leftarrow \gamma_i \cdot L \oplus R$
for $i \leftarrow 1$ **to** $m{-}1$ **do**
    $C[i] \leftarrow E_K(M[i] \oplus Z[i]) \oplus Z[i]$
$X[m] \leftarrow \mathrm{len}(M[m]) \oplus L{\cdot}x^{-1} \oplus Z[m]$
$Y[m] \leftarrow E_K(X[m])$
$C[m] \leftarrow Y[m] \oplus M[m]$
$C \leftarrow C[1] \ldots C[m]$
Checksum $\leftarrow M[1] \oplus \ldots \oplus M[m{-}1] \oplus \mathrm{C}[m]0^* \oplus Y[m]$
$T \leftarrow E_K(\text{Checksum} \oplus Z[m])[\text{first } \tau \text{ bits}]$
return $C \leftarrow C \parallel T$

Algorithm OCB.Dec$_K$ (N, M)

Partition C into $C[1] \ldots C[m]$ T

$L \leftarrow E_K(0^n)$
$R \leftarrow E_K(N \oplus \mathrm{L})$
for $i \leftarrow 1$ **to** m **do** $Z[i] \leftarrow \gamma_i \cdot L \oplus R$
for $i \leftarrow 1$ **to** $m{-}1$ **do**
    $M[i] \leftarrow E_K^{-1}(C[i] \oplus Z[i]) \oplus Z[i]$
$X[m] \leftarrow \mathrm{len}(C[m]) \oplus L{\cdot}x^{-1} \oplus Z[m]$
$Y[m] \leftarrow E_K(X[m])$
$M[m] \leftarrow Y[m] \oplus C[m]$
$M \leftarrow M[1] \ldots M[m]$
Checksum $\leftarrow M[1] \oplus \ldots \oplus M[m{-}1] \oplus \mathrm{C}[m]0^* \oplus Y[m]$
$T' \leftarrow E_K(\text{Checksum} \oplus Z[m])[\text{first } \tau \text{ bits}]$
if $T' = T$ **then return** M
    else return INVALID

Figure 54—OCB Encryption. The message to encrypt is M and the key is K. Message M is written as $M = M[1]\, M[2] \ldots M[m{-}1]\, M[m]$, where $m = \max\{1, \lceil |M|/n \rceil\}$ and $|M[1]| = |M[2]| = \ldots = |M[m{-}1]| = n$. Nonce N is a non-repeating value selected by the party that encrypts. It is sent along with ciphertext $C = C[1]\, C[2]\, C[3] \ldots C[m{-}1]\, C[m]$ T. The Checksum is $M[1] \oplus \ldots \oplus M[m{-}1] \oplus C[m]0^* \oplus Y[m]$. Offset $Z[1] = L \oplus R$ while, for $i \geq 2$, $Z[i] = Z[i{-}1] \oplus L(\mathrm{ntz}[i])$. String L is defined by applying E_K to a fixed string, 0^n. For $Y[m] \oplus M[m]$ and $Y[m] \oplus C[m]$, truncate $Y[m]$ if it is longer than the other operand. By $C[m]0^*$ we mean $C[m]$ padded on the right with 0-bits to get to length n. The function len represents the length of its argument as an n-bit string.

Encryption. To encrypt plaintext $M \in \{0, 1\}^*$ using key K and nonce $N \in \{0, 1\}^n$, obtaining a ciphertext C, do the following. Let $m \leftarrow \lceil |M|/n \rceil$. If $m = 0$ then let $m \leftarrow 1$. Let $M[1], \ldots, M[m]$ be strings such that $M[1] \ldots M[m] = M$ and $|M[i]| = n$ for $i \in [1..m\text{-}1]$. Let Offset $\leftarrow E_K(N \oplus L)$. Let Checksum $\leftarrow 0^n$. For $i \leftarrow 1$ to $m\text{-}1$, do the following: let Checksum $\leftarrow$ Checksum $\oplus M[i]$; let Offset $\leftarrow$ Offset $\oplus L(\mathrm{ntz}(i))$; let $C[i] \leftarrow E_K(M[i] \oplus \text{Offset}) \oplus \text{Offset}$. Let Offset $\leftarrow$ Offset $\oplus L(\mathrm{ntz}(m))$. Let $Y[m] \leftarrow E_K(\mathrm{len}(M[m]) \oplus L(\text{-}1) \oplus \text{Offset}$. Let $C[m] \leftarrow M[m]$ xored with the first $|M[m]|$ bits of $Y[m]$. Let Checksum $\leftarrow$ Checksum $\oplus Y[m] \oplus C[m]0^*$. Let T be the first τ bits of $E_K(\text{Checksum} \oplus \text{Offset})$. The ciphertext is $C = C[1] \ldots C[\mathrm{m}\text{-}1]\, C[m]\, T$. It must be communicated along with the nonce N.

Decryption. To decrypt ciphertext $C \in \{0, 1\}^*$ using key K and nonce $N \in \{0, 1\}^n$, obtaining a plaintext $M \in \{0, 1\}^*$ or an indication INVALID, do the following. If $|C| < \tau$ then return INVALID (the ciphertext has been rejected). Otherwise let C be the first $|C| - \tau$ bits of C and let T be the remaining τ bits. Let $m \leftarrow \lceil |C|/n \rceil$. If $m = 0$ then let $m = 1$. Let $C[1], \ldots, C[m]$ be strings such that $C[1] \ldots C[m] = C$ and $|C[i]| = n$ for $i \in [1..m\text{-}1]$. Let Offset $\leftarrow E_K(N \oplus L)$. Let Checksum $\leftarrow 0^n$. For $i \leftarrow 1$ to $m\text{-}1$, do the following: let Offset $\leftarrow$

Offset $\oplus$ L(ntz(i)); let $M[i] \leftarrow E_K^{-1}(C[i] \oplus \text{Offset}) \oplus \text{Offset}$; let Checksum $\leftarrow$ Checksum $\oplus$ $M[i]$. Let Offset $\leftarrow$ Offset $\oplus$ L(ntz(m)). Let $Y[m] \leftarrow E_K(\text{len}(C[m]) \oplus L(-1) \oplus \text{Offset})$. Let $M[m] \leftarrow C[m]$ xored with the first $|C[m]|$ bits of $Y[m]$. Let Checksum $\leftarrow$ Checksum $\oplus$ $Y[m]$ $\oplus$ $C[m]0^*$. Let T' be the first τ bits of E_K(Checksum $\oplus$ Offset). If $T \neq T'$ then return INVALID (the ciphertext has been rejected). Otherwise, the plaintext is $M = M[1] \ldots M[m-1]\ M[m]$.

F.6.2. OCB reference implementation

```
/*
 * ocb.h
 *
 * Author:  Ted Krovetz (tdk@acm.org)
 * History: 1 April 2000 - first release (TK) - version 0.9
 *
 * OCB-AES-n reference code based on NIST submission "OCB Mode"
 * (dated 1 April 2000), submitted by Phillip Rogaway, with
 * auxiliary submitters Mihir Bellare, John Black, and Ted Krovetz.
 *
 * This code is freely available, and may be modified as desired.
 * Please retain the authorship and change history.
 * Note that OCB mode itself is patent pending.
 *
 * This code is NOT optimized for speed; it is only
 * designed to clarify the algorithm and to provide a point
 * of comparison for other implementations.
 *
 * Limitiations:  Assumes a 4-byte integer and pointers are
 * 32-bit aligned. Acts on a byte string of less than 2^{36} - 16 bytes.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS
 * OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHORS OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
 * BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
 * OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
 * EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

#ifndef __OCB__H
#define __OCB__H

#ifndef AES_KEY_BITLEN
#define AES_KEY_BITLEN   128  /* Must be 128, 192, 256 */
#endif

#if ((AES_KEY_BITLEN != 128) && \
      (AES_KEY_BITLEN != 192) && \
      (AES_KEY_BITLEN != 256))
#error Bad -- AES_KEY_BITLEN must be one of 128, 192 or 256!!
#endif

/* Opaque forward declaration of key structure */
typedef struct _keystruct keystruct;

/*
 * "ocb_aes_init" optionally creates an ocb keystructure in memory
 * and then initializes it using the supplied "enc_key". "tag_len"
 * specifies the length of tags that will subsequently be generated
 * and verified. If "key" is NULL a new structure will be created, but
 * if "key" is non-NULL, then it is assumed that it points to a
 * previously allocated structure, and that structure is initialized.
```

```
 * "ocb_aes_init" returns a pointer to the initialized structure, or NULL
 * if an error occurred.
 */
keystruct *                      /* Init'd keystruct or NULL      */
ocb_aes_init(void *enc_key,      /* AES key                       */
      unsigned tag_len,         /* Length of tags to be used     */
      keystruct *key);          /* OCB key structure. NULL means */
                                 /* Allocate/init new, non-NULL   */
                                 /* means init existing structure */

/* "ocb_done deallocates a key structure and returns NULL */
keystruct *
ocb_done(keystruct *key);

/*
 * "ocb_aes_encrypt takes a key structure, four buffers and a length
 * parameter as input. "pt_len" bytes that are pointed to by "pt" are
 * encrypted and written to the buffer pointed to by "ct". A tag of
 * length "tag_len" (set in ocb_aes_init) is written to the "tag" buffer.
 * "nonce" must be a 16-byte buffer which changes for each new message
 * being encrypted. "ocb_aes_encrypt" always returns a value of 1.
 */
void
ocb_aes_encrypt(keystruct *key,    /* Initialized key struct           */
      void      *nonce,  /* 16-byte nonce                    */
      void      *pt,     /* Buffer for (incoming) plaintext  */
      unsigned   pt_len, /* Byte length of pt                */
      void      *ct,     /* Buffer for (outgoing) ciphertext */
      void      *tag);   /* Buffer for generated tag         */


/*
 * "ocb_aes_decrypt takes a key structure, four buffers and a length
 * parameter as input. "ct_len" bytes that are pointed to by "ct" are
 * decrypted and written to the buffer pointed to by "pt". A tag of
 * length "tag_len" (set in ocb_aes_init) is read from the "tag" buffer.
 * "nonce" must be a 16-byte buffer which changes for each new message
 * being encrypted. "ocb_aes_decrypt" returns 0 if the supplied
 * tag is not correct for the supplied message, otherwise 1 is returned
 * if the tag is correct.
 */
int                                /* Returns 0 iff tag is incorrect   */
ocb_aes_decrypt(keystruct *key,    /* Initialized key struct           */
      void      *nonce,  /* 16-byte nonce                    */
      void      *ct,     /* Buffer for (incoming) ciphertext */
      unsigned   ct_len, /* Byte length of ct                */
      void      *pt,     /* Buffer for (outgoing) plaintext  */
      void      *tag);   /* Tag to be verified               */

void
pmac_aes (keystruct *key,    /* Initialized key struct          */
      void      *in,     /* Buffer for (incoming) message    */
      unsigned   in_len, /* Byte length of message           */
      void      *tag);    /* 16-byte buffer for generated tag */

#endif /* __OCB__H */
```

```
/**
 * rijndael-alg-fst.h
 *
 * @version 3.0 (December 2000)
 *
 * Optimized ANSI C code for the Rijndael cipher (now AES)
 *
 * @author Vincent Rijmen <vincent.rijmen@esat.kuleuven.ac.be>
 * @author Antoon Bosselaers <antoon.bosselaers@esat.kuleuven.ac.be>
```

```
 * @author Paulo Barreto <paulo.barreto@terra.com.br>
 *
 * This code is hereby placed in the public domain.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHORS ''AS IS'' AND ANY EXPRESS
 * OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHORS OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
 * BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
 * OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
 * EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */
#ifndef __RIJNDAEL_ALG_FST_H
#define __RIJNDAEL_ALG_FST_H

#define MAXKC (256/32)
#define MAXKB (256/8)
#define MAXNR 14

typedef unsigned char      u8;
typedef unsigned short     u16;
typedef unsigned int       u32;

int rijndaelKeySetupEnc(
      u32 rk[/*4*(Nr + 1)*/], const u8 cipherKey[], int keyBits);
int rijndaelKeySetupDec(
      u32 rk[/*4*(Nr + 1)*/], const u8 cipherKey[], int keyBits);
void rijndaelEncrypt(
      const u32 rk[/*4*(Nr + 1)*/], int Nr, const u8 pt[16], u8 ct[16]);
void rijndaelDecrypt(
      const u32 rk[/*4*(Nr + 1)*/], int Nr, const u8 ct[16], u8 pt[16]);

#ifdef INTERMEDIATE_VALUE_KAT
void rijndaelEncryptRound(
      const u32 rk[/*4*(Nr + 1)*/], int Nr, u8 block[16], int rounds);
void rijndaelDecryptRound(
      const u32 rk[/*4*(Nr + 1)*/], int Nr, u8 block[16], int rounds);
#endif /* INTERMEDIATE_VALUE_KAT */

#endif /* __RIJNDAEL_ALG_FST_H */


/*
 * ocb.c
 *
 * Author:  Ted Krovetz (tdk@acm.org)
 * History: 1 April 2000 - first release (TK) - version 0.9
 *
 * OCB-AES-n reference code based on NIST submission "OCB Mode"
 * (dated 1 April 2000), submitted by Phillip Rogaway, with
 * auxiliary submitters Mihir Bellare, John Black, and Ted Krovetz.
 *
 * This code is freely available, and may be modified as desired.
 * Please retain the authorship and change history.
 * Note that OCB mode itself is patent pending.
 *
 * This code is NOT optimized for speed; it is only
 * designed to clarify the algorithm and to provide a point
 * of comparison for other implementations.
 *
 * Limitiations:  Assumes a 4-byte integer type and pointers that are
 * 32-bit aligned. Acts on a byte string of at most 2^36-16 bytes.
 *
```

```
 * Rijndael source available at www.esat.kuleuven.ac.be/~rijmen/rijndael/
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS
 * OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHORS OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
 * BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
 * OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
 * EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

#include "ocb.h"
#include "rijndael-alg-fst.h"
#include <stdlib.h>
#include <string.h>
#include <limits.h>

#if (INT_MAX != 0x7fffffff)
#error -- Assumes 4-byte int
#endif

/*
 * This implementation precomputes L(-1), L(0), L(1), L(PRE_COMP_BLOCKS),
 * where L(0) = L and L(-1) = L/x and L(i) = x*L(i) for i>0.
 * Normally, one would select PRE_COMP_BLOCKS to be a small number
 * (like 0-6) and compute any larger L(i) values "on the fly", when they
 * are needed.  This saves space in _keystruct and needn't adversely
 * impact running time.  But in this implementation, to keep things as
 * simple as possible, we compute all the L(i)-values we might ever see.
 */
#define PRE_COMP_BLOCKS 31     /* Must be between 0 and 31 */

#define AES_ROUNDS (AES_KEY_BITLEN / 32 + 6)

typedef unsigned char block[16];

struct _keystruct {
    unsigned rek[4*(AES_ROUNDS+1)]; /* AES encryption key */
    unsigned rdk[4*(AES_ROUNDS+1)]; /* AES decryption key */
    unsigned tag_len;               /* Sizeof tags to generate/validate */
    block L[PRE_COMP_BLOCKS+1];     /* Precomputed L(i) values, L[0] = L */
    block L_inv;                    /* Precomputed L/x value  */
};

/*************************************************************************
 * xor_block
 *************************************************************************/
static void xor_block(void *dst, void *src1, void *src2)
/*
 * 128-bit xor: *dst = *src1 xor *src2. Pointers must be 32-bit aligned
 */
{
      ((unsigned *)dst)[0] = ((unsigned *)src1)[0] ^ ((unsigned
             *)src2)[0];
      ((unsigned *)dst)[1] = ((unsigned *)src1)[1] ^ ((unsigned
             *)src2)[1];
      ((unsigned *)dst)[2] = ((unsigned *)src1)[2] ^ ((unsigned
             *)src2)[2];
      ((unsigned *)dst)[3] = ((unsigned *)src1)[3] ^ ((unsigned
             *)src2)[3];
}
```

```
/***********************************************************************
 * shift_left
 ***********************************************************************/
static void shift_left(unsigned char *x)
/*
 * 128-bit shift-left by 1 bit: *x <<= 1.
 */
{
      int i;
      for (i = 0; i < 15; i++) {
            x[i] = (x[i] << 1) | (x[i+1] & 0x80 ? 1 : 0);
      }
      x[15] = (x[15] << 1);
}

/**********************************************************************
 * shift_right
 ***********************************************************************/
static void shift_right(unsigned char *x)
/*
 * 128-bit shift-right by 1 bit:  *x >>= 1
 */
{
      int i;
      for (i = 15; i > 0; i--) {
            x[i] = (x[i] >> 1) | (x[i-1] & 1 ? 0x80u : 0);
      }
      x[0] = (x[0] >> 1);
}

/***********************************************************************
 * ntz
 ***********************************************************************/
static int ntz(unsigned i)
/*
 * Count the number of trailing zeroes in integer i.
 */
{
#if (MSC_VER && _M_IX86)  /* Only non-C sop */
      asm bsf eax, i
#elif (__GNUC__ && __i386__)
      int rval;
      asm volatile("bsf %1, %0" : "=r" (rval) : "g" (i));
      return rval;
#else
      int rval = 0;
      while ((i & 1) == 0) {
            i >>= 1;
            rval++;
      }
      return rval;
#endif
}

/***********************************************************************
 * ocb_aes_init
 ***********************************************************************/
keystruct *                         /* Init'd keystruct or NULL      */
ocb_aes_init(void      *enc_key,    /* AES key                       */
             unsigned   tag_len,    /* Length of tags to be used     */
             keystruct *key)        /* OCB key structure. NULL means */
                                    /* Allocate/init new, non-NULL   */
                                    /* means init existing structure */
{
      unsigned char tmp[16] = {0,};
      unsigned first_bit, last_bit, i;
```

```
        if (key == NULL)
            key = (keystruct *)malloc(sizeof(keystruct));
        if (key != NULL) {
            memset(key, 0, sizeof(keystruct));

            /* Initialize AES keys. (Note that if one is only going to
            encrypt, key->rdk can be eliminated */
            rijndaelKeySetupEnc(key->rek, (unsigned char *)enc_key,
                AES_KEY_BITLEN);
            rijndaelKeySetupDec(key->rdk, (unsigned char *)enc_key,
                AES_KEY_BITLEN);

            /* Precompute L[i]-values. L[0] is synonym of L */
            rijndaelEncrypt (key->rek, AES_ROUNDS, tmp, tmp);
            for (i = 0; i <= PRE_COMP_BLOCKS; i++) {
                memcpy(key->L + i, tmp, 16);   /* Copy tmp to L[i] */
                first_bit = tmp[0] & 0x80u;  /* multiply tmp by x */
                shift_left(tmp);
                if (first_bit)
                    tmp[15] ^= 0x87;
            }

            /* Precompute L_inv = L . x^{-1} */
            memcpy(tmp, key->L, 16);
            last_bit = tmp[15] & 0x01;
            shift_right(tmp);
            if (last_bit) {
                tmp[0] ^= 0x80;
                tmp[15] ^= 0x43;
            }
            memcpy(key->L_inv, tmp, 16);

            /* Set tag length used for this session */
            key->tag_len = tag_len;
        }

        return key;
    }

    /************************************************************************
     * ocb_aes_encrypt
     ************************************************************************/
    void
    ocb_aes_encrypt(keystruct *key,    /* Initialized key struct           */
                    void      *nonce,  /* 16-byte nonce                    */
                    void      *pt,     /* Buffer for (incoming) plaintext  */
                    unsigned   pt_len, /* Byte length of pt                */
                    void      *ct,     /* Buffer for (outgoing) ciphertext */
                    void      *tag)    /* Buffer for generated tag         */
    {
        unsigned i;                 /* Block counter                    */
        block tmp, tmp2;            /* temporary buffers                */
        block *pt_blk, *ct_blk;     /* block-typed aliases for pt / ct */
        block Offset;               /* Offset (Z[i]) for current block */
        block checksum;             /* Checksum for computing tag       */

        /*
         * Initializations
         */
        i = 1;                     /* Start with first block         */
        pt_blk = (block *)pt - 1;  /* These are adjusted so, e.g.,   */
        ct_blk = (block *)ct - 1;  /* pt_blk[1] refers to 1st block  */
        memset(checksum, 0, 16);   /* Zero the checksum              */

        /* Calculate R, aka Z[0] */
        xor_block(Offset, nonce, key->L);
        rijndaelEncrypt (key->rek, AES_ROUNDS, Offset, Offset);
```

```
        /*
         * Process blocks 1 .. m-1
         */
        while (pt_len > 16) {
                /* Update the Offset (Z[i] from Z[i-1]) */
                xor_block(Offset, key->L + ntz(i), Offset);

                /* xor the plaintext block with Z[i] */
                xor_block(tmp, Offset, pt_blk + i);

                /* Encipher the block */
                rijndaelEncrypt (key->rek, AES_ROUNDS, tmp, tmp);

                /* xor Z[i] again, writing result to ciphertext pointer */
                xor_block(ct_blk + i, Offset, tmp);

                /* Update checksum */
                xor_block(checksum, checksum, pt_blk + i);

                /* Update loop variables */
                pt_len -= 16;
                i++;
        }

        /*
         * Process block m
         */

        /* Update Offset (Z[m] from Z[m-1]) */
        xor_block(Offset, key->L + ntz(i), Offset);

        /* xor L . x^{-1} and Z[m] */
        xor_block(tmp, Offset, key->L_inv);

        /* Add in final block bit-length */
        tmp[15] ^= (pt_len << 3);

        rijndaelEncrypt (key->rek, AES_ROUNDS, tmp, tmp);

        /* xor 'pt' with block-cipher output, copy valid bytes to 'ct' */
        memcpy(tmp2, pt_blk + i, pt_len);
        xor_block(tmp2, tmp2, tmp);
        memcpy(ct_blk + i, tmp2, pt_len);

        /* Add to checksum the pt_len bytes of plaintext followed by */
        /* the last (16 - pt_len) bytes of block-cipher output */
        memcpy(tmp, pt_blk + i, pt_len);
        xor_block(checksum, checksum, tmp);

        /*
         * Calculate tag
         */
        xor_block(checksum, checksum, Offset);
        rijndaelEncrypt(key->rek, AES_ROUNDS, checksum, tmp);
        memcpy(tag, tmp, key->tag_len);
}

/************************************************************************
 * ocb_aes_decrypt
 ***********************************************************************/
int                                /* Returns 0 iff tag is incorrect   */
ocb_aes_decrypt(keystruct *key,    /* Initialized key struct           */
                void      *nonce,  /* 16-byte nonce                    */
                void      *ct,     /* Buffer for (incoming) ciphertext */
                unsigned   ct_len, /* Byte length of ct                */
                void      *pt,     /* Buffer for (outgoing) plaintext  */
```

```
                void      *tag)    /* Tag to be verified              */
    {
        unsigned i;                 /* Block counter                   */
        block tmp, tmp2;            /* temporary buffers               */
        block  *ct_blk, *pt_blk;    /* block-typed aliases for ct / pt */
        block Offset;               /* Offset (Z[i]) for current block */
        block checksum;             /* Checksum for computing tag      */

        /*
         * Initializations
         */
        i = 1;                     /* Start with first block          */
        ct_blk = (block *)ct - 1;  /* These are adjusted so, e.g.,    */
        pt_blk = (block *)pt - 1;  /* ct_blk[1] refers to 1st block   */

        /* Zero checksum */
        memset(checksum, 0, 16);

        /* Calculate R, aka Z[0] */
        xor_block(Offset, nonce, key->L);
        rijndaelEncrypt (key->rek, AES_ROUNDS, Offset, Offset);

        /*
         * Process blocks 1 .. m-1
         */
        while (ct_len > 16) {
              /* Update Offset (Z[i] from Z[i-1]) */
              xor_block(Offset, key->L + ntz(i), Offset);

              /* xor ciphertext block with Z[i] */
              xor_block(tmp, Offset, ct_blk + i);

              /* Decipher the next block-cipher block */
              rijndaelDecrypt (key->rdk, AES_ROUNDS, tmp, tmp);

              /* xor Z[i] again, writing result to plaintext pointer */
              xor_block(pt_blk + i, Offset, tmp);

              /* Update checksum */
              xor_block(checksum, checksum, pt_blk + i);

              /* Update loop variables */
              ct_len -= 16;
              i++;
        }

        /*
         * Process block m
         */

        /* Update Offset (Z[m] from Z[m-1]) */
        xor_block(Offset, key->L + ntz(i), Offset);

        /* xor L . x^{-1} and Z[m] */
        xor_block(tmp, Offset, key->L_inv);

        /* Add in final block bit-length */
        tmp[15] ^= (ct_len << 3);

        rijndaelEncrypt (key->rek, AES_ROUNDS, tmp, tmp);

        /* Form the final ciphertext block, C[m]  */
        memset(tmp2, 0, 16);
        memcpy(tmp2, ct_blk + i, ct_len);
        xor_block(tmp, tmp2, tmp);
        memcpy(pt_blk + i, tmp, ct_len);
```

```
                /* After xor above, tmp will have ct_len bytes of plaintext  */
                /* then (16 - ct_len) block-cipher bytes, perfect for chksum.  */
                xor_block(checksum, checksum, tmp);

                /*
                 * Calculate tag
                 */
                xor_block(checksum, checksum, Offset);
                rijndaelEncrypt(key->rek, AES_ROUNDS, checksum, tmp);
                return (memcmp(tag, tmp, key->tag_len) == 0 ? 1 : 0);
        }

        /**********************************************************************
         * ocb_done
         **********************************************************************/
        keystruct *ocb_done(keystruct *key)
        {
                if (key) {
                        memset(key, 0, sizeof(keystruct));
                        free(key);
                }
                return NULL;
        }
```

F.6.3 OCB test vectors

Test case OCB-AES-128-0B
Key 000102030405060708090a0b0c0d0e0f
Nonce 00000000000000000000000000000001
Plaintext <empty string>
Ciphertext <empty string>
Tag 15d37dd7c890d5d6acab927bc0dc60ee

Test case OCB-AES-128-3B
Key 000102030405060708090a0b0c0d0e0f
Nonce 00000000000000000000000000000001
Plaintext 000102
Ciphertext fcd37d
Tag 02254739a5e3565ae2dcd62c659746ba

Test case OCB-AES-128-16B
Key 000102030405060708090a0b0c0d0e0f
Nonce 00000000000000000000000000000001
Plaintext 000102030405060708090a0b0c0d0e0f
Ciphertext 37df8ce15b489bf31d0fc44da1faf6d6
Tag dfb763ebdb5f0e719c7b4161808004df

Test case OCB-AES-128-20B
Key 000102030405060708090a0b0c0d0e0f
Nonce 00000000000000000000000000000001
Plaintext 000102030405060708090a0b0c0d0e0f10111213
Ciphertext 01a075f0d815b1a4e9c881a1bcffc3eb7003eb55
Tag 753084144eb63b770b063c2e23cda0bb

Test case OCB-AES-128-32B
Key 000102030405060708090a0b0c0d0e0f
Nonce 00000000000000000000000000000001
Plaintext 000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
Ciphertext 01a075f0d815b1a4e9c881a1bcffc3eb4afcbb7fedc08ca8654c6d304d1612fa

Tag c14cbf2c1a1f1c3c137eadea1f2f2fcf

Test case OCB-AES-128-34B
Key 000102030405060708090a0b0c0d0e0f
Nonce 00000000000000000000000000000001
Plaintext 000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f2021
Ciphertext 01a075f0d815b1a4e9c881a1bcffc3ebd4903dd0025ba4aa837c74f121b0260fa95d
Tag cf8341bb10820ccf14bdec56b8d7d6ab

Test case OCB-AES-128-1000B
Key 000102030405060708090a0b0c0d0e0f
Nonce 00000000000000000000000000000001
Plaintext 00000000000000000000 ... 00000000000000000000 [1000 bytes]
Ciphertext 4c9b676705ff2df05503 ... 2f8d1496a60048e2b971 [1000 bytes]
Tag ab335f725475e33e90ab8c1e4891596d

F.7. CCM

F.7.1. CCM reference implementation

```
/*=================================================================
 * Proposed AES CTR/CBC-MAC mode test vector generation
 *
 * 11-02-001r2-I-AES-Encryption & Authentication
 * Using-CTR-Mode-with-CBC-MAC
 *
 * Author:  Doug Whiting, Hifn  (dwhiting@hifn.com)
 *
 * This code is released to the public domain, on an as-is basis.
 *
 *=================================================================
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <assert.h>

#include "aes_defs.h"       /* AES calling interface*/
#include "aes_vect.h"       /* NIST AES test vectors*/

typedef int BOOL;           /* boolean */

enum {
      BLK_SIZE   = 16,    /* # octets in an AES block */
      MAX_PACKET = 3*512, /* largest packet size */
      N_RESERVED = 0,     /* reserved nonce octet value */
      A_DATA     = 0x40,  /* the Adata bit in the flags */
      M_SHIFT    = 3,     /* how much to shift the 3-bit M field */
      L_SHIFT    = 0,     /* how much to shift the 3-bit L field */
      L_SIZE     = 2     /* size of the l(m) length field (in octets) */
};

union block {             /* AES cipher block */
      u32b x[BLK_SIZE/4]; /* access as 8-bit octets or 32-bit words */
      u08b b[BLK_SIZE];
};

struct packet {
```

```
        BOOL  encrypted; /* TRUE if encrypted */
        u08b  TA[6];     /* xmit address */
        int   micLength; /* # octets of MIC appended to plaintext (M) */
        int   clrCount;  /* # cleartext octets covered by MIC */
        u32b  pktNum[2]; /* unique packet sequence number (like WEP IV) */
        block key;       /* the encryption key (K) */
        int   length;    /* # octets in data[] */
        u08b  data[MAX_PACKET+2*BLK_SIZE]; /* packet contents */
};

struct {
        int   cnt;              /* how many words left in ct */
        block ptCntr;           /* the counter input */
        block ct;               /* the ciphertext (prng output) */
} prng;

/* return the 32-bit value read to be stored as a big-endian word */
u32b BigEndian(u32b x)
{
        static block b = {0,0,0,0};

        if (b.x[0] == 0)        /* first time, figure out endianness */
                b.x[0] = 0xFF000001;

        if (b.b[0] == 0xFF)     /* is this a big-endian CPU? */
                return x;       /* if so, just return x */

        if (b.b[0] != 0x01) /* not big-Endian; check it's little-Endian */
                assert(0);      /* if not, bomb! */

        /* little-endian: do the byte swapping */
        return (x >> 24) + (x << 24) +
                ((x >>  8) & 0x00FF00) + ((x <<  8) & 0xFF0000);
}

void InitRand(u32b seed)
{
        memset(prng.ptCntr.b,0,BLK_SIZE);
        prng.ptCntr.x[(BLK_SIZE/4)-1] = seed*17;
        prng.cnt = 0;  /* the pump is dry */
}

/* prng: does not use C rand(), so should be usable across platforms */
u32b Random32(void)
{
        if (prng.cnt == 0) { /* use whatever key is currently defined */
                prng.cnt = BLK_SIZE/4;
                prng.ptCntr.x[0]++;
                if (prng.ptCntr.x[0] == 0) /* ripple carry? */
                        prng.ptCntr.x[1]++;  /* stop at 64 bits */
                AES_Encrypt(prng.ptCntr.x, prng.ct.x);
        }
        --prng.cnt;
        return BigEndian(prng.ct.x[prng.cnt]);
}

/* display a block */
void ShowBlock(
        const block *blk,
        const char *prefix,
        const char *suffix,
        int a)
{
        int i, blkSize = BLK_SIZE;
        printf(prefix,a);
        if (suffix == NULL) {
                suffix = "\n";
```

```
                blkSize = a;
            }
            for (i = 0; i < blkSize; i++)
                printf("%02X%s", blk->b[i], ((i&3)==3) ? "  ":" ");
            printf (suffix);
    }

    void ShowAddr(const packet *p)
    {
            int i;

            printf("      TA = ");
            for (i = 0; i < 6 ; i++)
                printf("%02X%s",p->TA[i],(i==3)?"  ":" ");
            printf("  48-bit pktNum = %04X.%08X\n",p->pktNum[1],p->pktNum[0]);
    }

    /* display a packet */
    void ShowPacket(const packet *p, const char *pComment, int a)
    {
            int i;

            printf("Total packet length = %4d. ", p->length);
            printf(pComment, a);
            if (p->encrypted)
                printf("[Encrypted]");
            for (i = 0; i < p->length; i++) {
                if ((i & 15) == 0)
                    printf("\n%11s","");
                printf("%02X%s", p->data[i], ((i&3)==3) ? "  ":" ");
            }
            printf("\n");
    }

    /* make sure that encrypt/decrypt work according to NIST vectors */
    void Validate_NIST_AES_Vectors(int verbose)
    {
            int i;
            block key,pt,ct,rt;

            printf("AES KAT Vectors:\n");   /* known-answer tests */
            /* variable text (fixed-key) tests */
            memcpy(key.b,VT_key,BLK_SIZE);
            AES_SetKey(key.x,BLK_SIZE*8);
            for (i = 0; i < sizeof(VT_pt_ct_pairs); i += 2 * BLK_SIZE) {
                memcpy(pt.b, VT_pt_ct_pairs+i, BLK_SIZE);
                AES_Encrypt(pt.x, ct.x);
                if (memcmp(ct.x, VT_pt_ct_pairs+i+BLK_SIZE, BLK_SIZE)) {
                    printf("Vector miscompare at VT test #%d", i);
                    exit(1);
                }
                AES_Decrypt(ct.x, rt.x); /* sanity check on decrypt */
                if (memcmp(pt.b, rt.b, BLK_SIZE)) {
                    printf("Decrypt miscompare at VT test #%d", i);
                    exit(1);
                }
                if (verbose) { /* only do a little if we're "debugging" */
                    printf("\n");
                    break;
                } else if (i==0) {  /* display the first vector */
                    ShowBlock(&key,"Key:        ","\n",0);
                    ShowBlock(&pt ,"PT:         ","\n",0);
                    ShowBlock(&ct ,"CT:         ","\n\n",0);
                }
            }

            /* variable key (fixed-text) tests */
```

```
        memcpy(pt.b, VK_pt, BLK_SIZE);
        for (i = 0; i < sizeof(VK_key_ct_pairs); i += 2*BLK_SIZE) {
                memcpy(key.b, VK_key_ct_pairs+i, BLK_SIZE);
                AES_SetKey(key.x, BLK_SIZE*8);
                AES_Encrypt(pt.x, ct.x);
                if (memcmp(ct.x, VK_key_ct_pairs+i+BLK_SIZE, BLK_SIZE)) {
                        printf("Vector miscompare at VK test #%d", i);
                        exit(1);
                }
                AES_Decrypt(ct.x, rt.x); /* sanity check on decrypt */
                if (memcmp(pt.b, rt.b, BLK_SIZE)) {
                        printf("Decrypt miscompare at VK test #%d",i);
                        exit(1);
                }
                if (verbose) {  /* only do a little if we're "debugging" */
                        printf("\n");
                        break;
                } else if (i==0) {  /*  display the first vector */
                        ShowBlock(&key, "Key:        ", "\n", 0);
                        ShowBlock(&pt , "PT:         ", "\n", 0);
                        ShowBlock(&ct , "CT:         ", "\n\n", 0);
                }
        }
        printf("NIST AES Vectors: OK\n"); /* ok if we got here */
}

/* assumes AES_SetKey is called elsewhere */
void Generate_CTR_CBC_Vector(packet *p, int verbose)
{
        int i, j, len, needPad, blkNum;
        block   m, x, T;
        assert(p->length >= p->clrCount && p->length <= MAX_PACKET);
        assert(p->micLength > 0 && p->micLength <= BLK_SIZE);
        len = p->length - p->clrCount;  /* l(m) */

        ShowPacket(p,"[Input (%d cleartext header octets)]", p->clrCount);

        /* ---- generate the first AES block for CBC-MAC */
        m.b[ 0] = (u08b) (((p->clrCount)?A_DATA:0) +
                (((p->micLength-2)/2 << M_SHIFT)) +
                ((L_SIZE-1) << L_SHIFT));  /* flags octet */
        m.b[ 1] = N_RESERVED;  /* reserved nonce octet  */
        m.b[ 2] = (u08b) (p->pktNum[1] >> 8) & 0xFF; /* 48 bit pkt # */
        m.b[ 3] = (u08b)  p->pktNum[1]       & 0xFF;
        m.b[ 4] = (u08b) (p->pktNum[0] >>24) & 0xFF;
        m.b[ 5] = (u08b) (p->pktNum[0] >>16) & 0xFF;
        m.b[ 6] = (u08b) (p->pktNum[0] >> 8) & 0xFF;
        m.b[ 7] = (u08b)  p->pktNum[0]       & 0xFF;
        m.b[ 8] = p->TA[0];  /* 48 bit Transmit Addr */
        m.b[ 9] = p->TA[1];
        m.b[10] = p->TA[2];
        m.b[11] = p->TA[3];
        m.b[12] = p->TA[4];
        m.b[13] = p->TA[5];
        m.b[14] = (len >> 8) & 0xFF;  /* l(m) field */
        m.b[15] = len        & 0xFF;

        /*---- compute the CBC-MAC tag (MIC) */
        AES_Encrypt(m.x, x.x);  /* produce the CBC IV */
        ShowBlock(&m,"CBC IV in: ", "\n", 0);
        if (verbose)
                ShowBlock(&x, "CBC IV out:", "\n", 0);
        j = 0;    /* j = octet counter inside the block */
        if (p->clrCount) {  /* is there a header? */
                /* if so, "insert" length field: l(a) */
                assert(p->clrCount < 0xFFF0);
                        /* [don't handle larger cases (yet)] */
```

```
                x.b[j++] ^= (p->clrCount >> 8) & 0xFF;
                x.b[j++] ^=  p->clrCount       & 0xFF;
        }
        for (i = blkNum = 0; i < p->length; i++) { /* CBC-MAC */
                x.b[j++] ^= p->data[i]; /* perform the CBC xor */
                needPad = (i == p->clrCount-1) || (i == p->length-1);
                if ((j == BLK_SIZE) || needPad) {
                        /* full block, or hit pad boundary */
                        if (verbose)
                                ShowBlock(&x, "After xor: ",
                                        (i >= p->clrCount) ? " [msg]\n" : "
                                        [hdr]\n",blkNum);
                        AES_Encrypt(x.x, x.x);  /* encrypt in place */
                        if (verbose)
                                ShowBlock(&x, "After AES: ", "\n", blkNum);
                        blkNum++;  /* count the blocks */
                        j = 0;  /* the block is now empty */
                }
        }
        memcpy(T.b,x.b,p->micLength);       // save the MIC tag
        ShowBlock(&T,"MIC tag  : ",NULL,p->micLength);

        /* ---- encrypt the data packet using CTR mode */
        m.b[0] &= ~(A_DATA | (7<<M_SHIFT));
                /* clear flag fields for counter mode */
        for (i=blkNum=0;i+p->clrCount < p->length;i++) {
                if ((i % BLK_SIZE) == 0) {
                        /* generate new keystream block */
                        blkNum++;  /* start data with block #1 */
                        m.b[14] = blkNum/256;
                        m.b[15] = blkNum%256;
                        AES_Encrypt(m.x, x.x); /* encrypt the counter */
                        if (verbose && i==0)
                                ShowBlock(&m,"CTR Start: ","\n",0);
                        if (verbose)
                                ShowBlock(&x,"CTR[%04X]: " ,"\n",blkNum);
                }
                /* merge in the keystream */
                p->data[i+p->clrCount] ^= x.b[i % BLK_SIZE];
        }

        /* ---- truncate, encrypt, and append MIC to packet */
        m.b[14] = m.b[15] = 0; /* use block counter value zero for tag */
        AES_Encrypt(m.x, x.x); /* encrypt the counter */
        if (verbose)
                ShowBlock(&x,"CTR[MIC ]: " ,NULL,p->micLength);
        for (i = 0; i < p->micLength; i++)
                p->data[p->length+i] = T.b[i] ^ x.b[i];
        p->length += p->micLength; /* adjust pkt length accordingly */

        p->encrypted = 1;
        ShowPacket(p,"",0); /* show the final encrypted packet */
}

int main(int argc,char *argv[])
{
        int i, j, k, len, pktNum, seed;
        packet p;

        seed = (argc > 1) ? atoi(argv[1]) : (int) time(NULL);
        InitRand(seed);
        printf("%s C compiler [%s %s].\nRandom seed = %d\n",
                COMPILER_ID,__DATE__,__TIME__,seed);

        /* 1st, make sure that our AES code matches NIST KAT vectors */
        Validate_NIST_AES_Vectors(_VERBOSE_);
```

```
        /* generate CTR-CBC vectors for various parameter settings */
        for (k = pktNum = 0; k < 2; k++) {
                /* k==1 => random vectors.
                   k==0 => "visually simple" vectors */
                for (i = 0; i < BLK_SIZE  ; i++)
                        p.key.b[i] =
                                k) ? (u08b) Random32() & 0xFF : i + 0xC0;
                for (i = 0; i < 6; i++)
                        p.TA[i] = (k) ? (u08b) Random32() & 0xFF : i + 0xA0;
                        AES_SetKey(p.key.x, BLK_SIZE*8);
                                /* run key schedule */

                /* now generate the vectors */
                for (p.micLength  = 8;p.micLength  <12;p.micLength+=2)
                for (p.clrCount   = 8;p.clrCount   <16;p.clrCount+=4)
                for (len          =32;len          <64;len*=2)
                for (i            =-1;i            < 2;i++) {
                        p.pktNum[0] = (k) ? Random32() :
                                pktNum*0x01010101 + 0x03020100;
                        p.pktNum[1] = (k) ? Random32() & 0xFFFF : 0;
                                /* 48-bit IV */
                        p.length    = len+i;  /* len+i is packet length */
                        p.encrypted = 0;
                        assert(p.length <= MAX_PACKET);
                        for (j = 0; j < p.length; j++) /* random pkt */
                                p.data[j]= (k) ? (u08b) Random32() & 0xFF : j;
                        pktNum++;
                        printf("========== Packet Vector #%d
                        =============\n",pktNum);
                        ShowBlock(&p.key ,"AES Key:   ","\n",0);
                        ShowAddr (&p);
                        Generate_CTR_CBC_Vector(&p,1);
                }
        }
        return 0;
}
```

F.7.2. CCM test vectors

The test vectors included in this annex cover the generic CCM mode, not the conventions for 802.11i.

```
=============== Packet Vector #1 ==================
AES Key:   C0 C1 C2 C3  C4 C5 C6 C7  C8 C9 CA CB  CC CD CE CF
      TA = A0 A1 A2 A3  A4 A5   48-bit pktNum = 0000.03020100
Total packet length =   31. [Input (8 cleartext header octets)]
           00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
           10 11 12 13  14 15 16 17  18 19 1A 1B  1C 1D 1E
CBC IV in: 59 00 00 00  03 02 01 00  A0 A1 A2 A3  A4 A5 00 17
CBC IV out:EB 9D 55 47  73 09 55 AB  23 1E 0A 2D  FE 4B 90 D6
After xor: EB 95 55 46  71 0A 51 AE  25 19 0A 2D  FE 4B 90 D6   [hdr]
After AES: CD B6 41 1E  3C DC 9B 4F  5D 92 58 B6  9E E7 F0 91
After xor: C5 BF 4B 15  30 D1 95 40  4D 83 4A A5  8A F2 E6 86   [msg]
After AES: 9C 38 40 5E  A0 3C 1B C9  04 B5 8B 40  C7 6C A2 EB
After xor: 84 21 5A 45  BC 21 05 C9  04 B5 8B 40  C7 6C A2 EB   [msg]
After AES: 2D C6 97 E4  11 CA 83 A8  60 C2 C4 06  CC AA 54 2F
MIC tag  : 2D C6 97 E4  11 CA 83 A8
CTR Start: 01 00 00 00  03 02 01 00  A0 A1 A2 A3  A4 A5 00 01
CTR[0001]: 50 85 9D 91  6D CB 6D DD  E0 77 C2 D1  D4 EC 9F 97
CTR[0002]: 75 46 71 7A  C6 DE 9A FF  64 0C 9C 06  DE 6D 0D 8F
CTR[MIC ]: 3A 2E 46 C8  EC 33 A5 48
Total packet length =   39. [Encrypted]
           00 01 02 03  04 05 06 07  58 8C 97 9A  61 C6 63 D2
           F0 66 D0 C2  C0 F9 89 80  6D 5F 6B 61  DA C3 84 17
           E8 D1 2C FD  F9 26 E0
```

```
=============== Packet Vector #2 ==================
AES Key:   C0 C1 C2 C3  C4 C5 C6 C7  C8 C9 CA CB  CC CD CE CF
      TA = A0 A1 A2 A3  A4 A5   48-bit pktNum = 0000.04030201
Total packet length =   32. [Input (8 cleartext header octets)]
           00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
           10 11 12 13  14 15 16 17  18 19 1A 1B  1C 1D 1E 1F
CBC IV in: 59 00 00 00  04 03 02 01  A0 A1 A2 A3  A4 A5 00 18
CBC IV out:F0 C2 54 D3  CA 03 E2 39  70 BD 24 A8  4C 39 9E 77
After xor: F0 CA 54 D2  C8 00 E6 3C  76 BA 24 A8  4C 39 9E 77   [hdr]
After AES: 48 DE 8B 86  28 EA 4A 40  00 AA 42 C2  95 BF 4A 8C
After xor: 40 D7 81 8D  24 E7 44 4F  10 BB 50 D1  81 AA 5C 9B   [msg]
After AES: 0F 89 FF BC  A6 2B C2 4F  13 21 5F 16  87 96 AA 33
After xor: 17 90 E5 A7  BA 36 DC 50  13 21 5F 16  87 96 AA 33   [msg]
After AES: F7 B9 05 6A  86 92 6C F3  FB 16 3D C4  99 EF AA 11
MIC tag  : F7 B9 05 6A  86 92 6C F3
CTR Start: 01 00 00 00  04 03 02 01  A0 A1 A2 A3  A4 A5 00 01
CTR[0001]: 7A C0 10 3D  ED 38 F6 C0  39 0D BA 87  1C 49 91 F4
CTR[0002]: D4 0C DE 22  D5 F9 24 24  F7 BE 9A 56  9D A7 9F 51
CTR[MIC ]: 57 28 D0 04  96 D2 65 E5
Total packet length =   40. [Encrypted]
           00 01 02 03  04 05 06 07  72 C9 1A 36  E1 35 F8 CF
           29 1C A8 94  08 5C 87 E3  CC 15 C4 39  C9 E4 3A 3B
           A0 91 D5 6E  10 40 09 16

=============== Packet Vector #3 ==================
AES Key:   C0 C1 C2 C3  C4 C5 C6 C7  C8 C9 CA CB  CC CD CE CF
      TA = A0 A1 A2 A3  A4 A5   48-bit pktNum = 0000.05040302
Total packet length =   33. [Input (8 cleartext header octets)]
           00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
           10 11 12 13  14 15 16 17  18 19 1A 1B  1C 1D 1E 1F
           20
CBC IV in: 59 00 00 00  05 04 03 02  A0 A1 A2 A3  A4 A5 00 19
CBC IV out:6F 8A 12 F7  BF 8D 4D C5  A1 19 6E 95  DF F0 B4 27
After xor: 6F 82 12 F6  BD 8E 49 C0  A7 1E 6E 95  DF F0 B4 27   [hdr]
After AES: 37 E9 B7 8C  C2 20 17 E7  33 80 43 0C  BE F4 28 24
After xor: 3F E0 BD 87  CE 2D 19 E8  23 91 51 1F  AA E1 3E 33   [msg]
After AES: 90 CA 05 13  9F 4D 4E CF  22 6F E9 81  C5 9E 2D 40
After xor: 88 D3 1F 08  83 50 50 D0  02 6F E9 81  C5 9E 2D 40   [msg]
After AES: 73 B4 67 75  C0 26 DE AA  41 03 97 D6  70 FE 5F B0
MIC tag  : 73 B4 67 75  C0 26 DE AA
CTR Start: 01 00 00 00  05 04 03 02  A0 A1 A2 A3  A4 A5 00 01
CTR[0001]: 59 B8 EF FF  46 14 73 12  B4 7A 1D 9D  39 3D 3C FF
CTR[0002]: 69 F1 22 A0  78 C7 9B 89  77 89 4C 99  97 5C 23 78
CTR[MIC ]: 39 6E C0 1A  7D B9 6E 6F
Total packet length =   41. [Encrypted]
           00 01 02 03  04 05 06 07  51 B1 E5 F4  4A 19 7D 1D
           A4 6B 0F 8E  2D 28 2A E8  71 E8 38 BB  64 DA 85 96
           57 4A DA A7  6F BD 9F B0  C5

=============== Packet Vector #4 ==================
AES Key:   C0 C1 C2 C3  C4 C5 C6 C7  C8 C9 CA CB  CC CD CE CF
      TA = A0 A1 A2 A3  A4 A5   48-bit pktNum = 0000.06050403
Total packet length =   31. [Input (12 cleartext header octets)]
           00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
           10 11 12 13  14 15 16 17  18 19 1A 1B  1C 1D 1E
CBC IV in: 59 00 00 00  06 05 04 03  A0 A1 A2 A3  A4 A5 00 13
CBC IV out:06 65 2C 60  0E F5 89 63  CA C3 25 A9  CD 3E 2B E1
After xor: 06 69 2C 61  0C F6 8D 66  CC C4 2D A0  C7 35 2B E1   [hdr]
After AES: A0 75 09 AC  15 C2 58 86  04 2F 80 60  54 FE A6 86
After xor: AC 78 07 A3  05 D3 4A 95  10 3A 96 77  4C E7 BC 9D   [msg]
After AES: 64 4C 09 90  D9 1B 83 E9  AB 4B 8E ED  06 6F F5 BF
After xor: 78 51 17 90  D9 1B 83 E9  AB 4B 8E ED  06 6F F5 BF   [msg]
After AES: 4B 4F 4B 39  B5 93 E6 BF  B0 B2 C2 B7  0F 29 CD 7A
MIC tag  : 4B 4F 4B 39  B5 93 E6 BF
CTR Start: 01 00 00 00  06 05 04 03  A0 A1 A2 A3  A4 A5 00 01
CTR[0001]: AE 81 66 6A  83 8B 88 6A  EE BF 4A 5B  32 84 50 8A
CTR[0002]: D1 B1 92 06  AC 93 9E 2F  B6 DD CE 10  A7 74 FD 8D
```

```
CTR[MIC ]: DD 87 2A 80  7C 75 F8 4E
Total packet length =   39. [Encrypted]
           00 01 02 03  04 05 06 07  08 09 0A 0B  A2 8C 68 65
           93 9A 9A 79  FA AA 5C 4C  2A 9D 4A 91  CD AC 8C 96
           C8 61 B9 C9  E6 1E F1

=============== Packet Vector #5 ==================
AES Key:   C0 C1 C2 C3  C4 C5 C6 C7  C8 C9 CA CB  CC CD CE CF
      TA = A0 A1 A2 A3  A4 A5   48-bit pktNum = 0000.07060504
Total packet length =   32. [Input (12 cleartext header octets)]
           00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
           10 11 12 13  14 15 16 17  18 19 1A 1B  1C 1D 1E 1F
CBC IV in: 59 00 00 00  07 06 05 04  A0 A1 A2 A3  A4 A5 00 14
CBC IV out:00 4C 50 95  45 80 3C 48  51 CD E1 3B  56 C8 9A 85
After xor: 00 40 50 94  47 83 38 4D  57 CA E9 32  5C C3 9A 85   [hdr]
After AES: E2 B8 F7 CE  49 B2 21 72  84 A8 EA 84  FA AD 67 5C
After xor: EE B5 F9 C1  59 A3 33 61  90 BD FC 93  E2 B4 7D 47   [msg]
After AES: 3E FB 36 72  25 DB 11 01  D3 C2 2F 0E  CA FF 44 F3
After xor: 22 E6 28 6D  25 DB 11 01  D3 C2 2F 0E  CA FF 44 F3   [msg]
After AES: 48 B9 E8 82  55 05 4A B5  49 0A 95 F9  34 9B 4B 5E
MIC tag  : 48 B9 E8 82  55 05 4A B5
CTR Start: 01 00 00 00  07 06 05 04  A0 A1 A2 A3  A4 A5 00 01
CTR[0001]: D0 FC F5 74  4D 8F 31 E8  89 5B 05 05  4B 7C 90 C3
CTR[0002]: 72 A0 D4 21  9F 0D E1 D4  04 83 BC 2D  3D 0C FC 2A
CTR[MIC ]: 19 51 D7 85  28 99 67 26
Total packet length =   40. [Encrypted]
           00 01 02 03  04 05 06 07  08 09 0A 0B  DC F1 FB 7B
           5D 9E 23 FB  9D 4E 13 12  53 65 8A D8  6E BD CA 3E
           51 E8 3F 07  7D 9C 2D 93

=============== Packet Vector #6 ==================
AES Key:   C0 C1 C2 C3  C4 C5 C6 C7  C8 C9 CA CB  CC CD CE CF
      TA = A0 A1 A2 A3  A4 A5   48-bit pktNum = 0000.08070605
Total packet length =   33. [Input (12 cleartext header octets)]
           00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
           10 11 12 13  14 15 16 17  18 19 1A 1B  1C 1D 1E 1F
           20
CBC IV in: 59 00 00 00  08 07 06 05  A0 A1 A2 A3  A4 A5 00 15
CBC IV out:04 72 DA 4C  6F F6 0A 63  06 52 1A 06  04 80 CD E5
After xor: 04 7E DA 4D  6D F5 0E 66  00 55 12 0F  0E 8B CD E5   [hdr]
After AES: 64 4C 36 A5  A2 27 37 62  0B 89 F1 D7  BF F2 73 D4
After xor: 68 41 38 AA  B2 36 25 71  1F 9C E7 C0  A7 EB 69 CF   [msg]
After AES: 41 E1 19 CD  19 24 CE 77  F1 2F A6 60  C1 6E BB 4E
After xor: 5D FC 07 D2  39 24 CE 77  F1 2F A6 60  C1 6E BB 4E   [msg]
After AES: A5 27 D8 15  6A C3 59 BF  1C B8 86 E6  2F 29 91 29
MIC tag  : A5 27 D8 15  6A C3 59 BF
CTR Start: 01 00 00 00  08 07 06 05  A0 A1 A2 A3  A4 A5 00 01
CTR[0001]: 63 CC BE 1E  E0 17 44 98  45 64 B2 3A  8D 24 5C 80
CTR[0002]: 39 6D BA A2  A7 D2 CB D4  B5 E1 7C 10  79 45 BB C0
CTR[MIC ]: E5 7D DC 56  C6 52 92 2B
Total packet length =   41. [Encrypted]
           00 01 02 03  04 05 06 07  08 09 0A 0B  6F C1 B0 11
           F0 06 56 8B  51 71 A4 2D  95 3D 46 9B  25 70 A4 BD
           87 40 5A 04  43 AC 91 CB  94

=============== Packet Vector #7 ==================
AES Key:   C0 C1 C2 C3  C4 C5 C6 C7  C8 C9 CA CB  CC CD CE CF
      TA = A0 A1 A2 A3  A4 A5   48-bit pktNum = 0000.09080706
Total packet length =   31. [Input (8 cleartext header octets)]
           00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
           10 11 12 13  14 15 16 17  18 19 1A 1B  1C 1D 1E
CBC IV in: 61 00 00 00  09 08 07 06  A0 A1 A2 A3  A4 A5 00 17
CBC IV out:60 06 C5 72  DA 23 9C BF  A0 5B 0A DE  D2 CD A8 1E
After xor: 60 0E C5 73  D8 20 98 BA  A6 5C 0A DE  D2 CD A8 1E   [hdr]
After AES: 41 7D E2 AE  94 E2 EA D9  00 FC 44 FC  D0 69 52 27
After xor: 49 74 E8 A5  98 EF E4 D6  10 ED 56 EF  C4 7C 44 30   [msg]
After AES: 2A 6C 42 CA  49 D7 C7 01  C5 7D 59 FF  87 16 49 0E
```

```
After xor: 32 75 58 D1  55 CA D9 01  C5 7D 59 FF  87 16 49 0E   [msg]
After AES: 89 8B D6 45  4E 27 20 BB  D2 7E F3 15  7A 7C 90 B2
MIC tag  : 89 8B D6 45  4E 27 20 BB  D2 7E
CTR Start: 01 00 00 00  09 08 07 06  A0 A1 A2 A3  A4 A5 00 01
CTR[0001]: 09 3C DB B9  C5 52 4F DA  C1 C5 EC D2  91 C4 70 AF
CTR[0002]: 11 57 83 86  E2 C4 72 B4  8E CC 8A AD  AB 77 6F CB
CTR[MIC ]: 8D 07 80 25  62 B0 8C 00  A6 EE
Total packet length =   41. [Encrypted]
           00 01 02 03  04 05 06 07  01 35 D1 B2  C9 5F 41 D5
           D1 D4 FE C1  85 D1 66 B8  09 4E 99 9D  FE D9 6C 04
           8C 56 60 2C  97 AC BB 74  90

=============== Packet Vector #8 ==================
AES Key:   C0 C1 C2 C3  C4 C5 C6 C7  C8 C9 CA CB  CC CD CE CF
      TA = A0 A1 A2 A3  A4 A5   48-bit pktNum = 0000.0A090807
Total packet length =   32. [Input (8 cleartext header octets)]
           00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
           10 11 12 13  14 15 16 17  18 19 1A 1B  1C 1D 1E 1F
CBC IV in: 61 00 00 00  0A 09 08 07  A0 A1 A2 A3  A4 A5 00 18
CBC IV out:63 A3 FA E4  6C 79 F3 FA  78 38 B8 A2  80 36 B6 0B
After xor: 63 AB FA E5  6E 7A F7 FF  7E 3F B8 A2  80 36 B6 0B   [hdr]
After AES: 1C 99 1A 3D  B7 60 79 27  34 40 79 1F  AD 8B 5B 02
After xor: 14 90 10 36  BB 6D 77 28  24 51 6B 0C  B9 9E 4D 15   [msg]
After AES: 14 19 E8 E8  CB BE 75 58  E1 E3 BE 4B  6C 9F 82 E3
After xor: 0C 00 F2 F3  D7 A3 6B 47  E1 E3 BE 4B  6C 9F 82 E3   [msg]
After AES: E0 16 E8 1C  7F 7B 8A 38  A5 38 F2 CB  5B B6 C1 F2
MIC tag  : E0 16 E8 1C  7F 7B 8A 38  A5 38
CTR Start: 01 00 00 00  0A 09 08 07  A0 A1 A2 A3  A4 A5 00 01
CTR[0001]: 73 7C 33 91  CC 8E 13 DD  E0 AA C5 4B  6D B7 EB 98
CTR[0002]: 74 B7 71 77  C5 AA C5 3B  04 A4 F8 70  8E 92 EB 2B
CTR[MIC ]: 21 6D AC 2F  8B 4F 1C 07  91 8C
Total packet length =   42. [Encrypted]
           00 01 02 03  04 05 06 07  7B 75 39 9A  C0 83 1D D2
           F0 BB D7 58  79 A2 FD 8F  6C AE 6B 6C  D9 B7 DB 24
           C1 7B 44 33  F4 34 96 3F  34 B4

=============== Packet Vector #9 ==================
AES Key:   C0 C1 C2 C3  C4 C5 C6 C7  C8 C9 CA CB  CC CD CE CF
      TA = A0 A1 A2 A3  A4 A5   48-bit pktNum = 0000.0B0A0908
Total packet length =   33. [Input (8 cleartext header octets)]
           00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
           10 11 12 13  14 15 16 17  18 19 1A 1B  1C 1D 1E 1F
           20
CBC IV in: 61 00 00 00  0B 0A 09 08  A0 A1 A2 A3  A4 A5 00 19
CBC IV out:4F 2C 86 11  1E 08 2A DD  6B 44 21 3A  B5 13 13 16
After xor: 4F 24 86 10  1C 0B 2E D8  6D 43 21 3A  B5 13 13 16   [hdr]
After AES: F6 EC 56 87  3C 57 12 DC  9C C5 3C A8  D4 D1 ED 0A
After xor: FE E5 5C 8C  30 5A 1C D3  8C D4 2E BB  C0 C4 FB 1D   [msg]
After AES: 17 C1 80 A5  31 53 D4 C3  03 85 0C 95  65 80 34 52
After xor: 0F D8 9A BE  2D 4E CA DC  23 85 0C 95  65 80 34 52   [msg]
After AES: 46 A1 F6 E2  B1 6E 75 F8  1C F5 6B 1A  80 04 44 1B
MIC tag  : 46 A1 F6 E2  B1 6E 75 F8  1C F5
CTR Start: 01 00 00 00  0B 0A 09 08  A0 A1 A2 A3  A4 A5 00 01
CTR[0001]: 8A 5A 10 6B  C0 29 9A 55  5B 93 6B 0B  0E A0 DE 5A
CTR[0002]: EA 05 FD E2  AB 22 5C FE  B7 73 12 CB  88 D9 A5 4A
CTR[MIC ]: AC 3D F1 07  DA 30 C4 86  43 BB
Total packet length =   43. [Encrypted]
           00 01 02 03  04 05 06 07  82 53 1A 60  CC 24 94 5A
           4B 82 79 18  1A B5 C8 4D  F2 1C E7 F9  B7 3F 42 E1
           97 EA 9C 07  E5 6B 5E B1  7E 5F 4E

=============== Packet Vector #10 ==================
AES Key:   C0 C1 C2 C3  C4 C5 C6 C7  C8 C9 CA CB  CC CD CE CF
      TA = A0 A1 A2 A3  A4 A5   48-bit pktNum = 0000.0C0B0A09
Total packet length =   31. [Input (12 cleartext header octets)]
           00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
           10 11 12 13  14 15 16 17  18 19 1A 1B  1C 1D 1E
```

```
CBC IV in: 61 00 00 00  0C 0B 0A 09  A0 A1 A2 A3  A4 A5 00 13
CBC IV out:7F B8 0A 32  E9 80 57 46  EC 31 6C 3A  B2 A2 EB 5D
After xor: 7F B4 0A 33  EB 83 53 43  EA 36 64 33  B8 A9 EB 5D   [hdr]
After AES: 7E 96 96 BF  F1 56 D6 A8  6E AC F5 7B  7F 23 47 5A
After xor: 72 9B 98 B0  E1 47 C4 BB  7A B9 E3 6C  67 3A 5D 41   [msg]
After AES: 8B 4A EE 42  04 24 8A 59  FA CC 88 66  57 66 DD 72
After xor: 97 57 F0 42  04 24 8A 59  FA CC 88 66  57 66 DD 72   [msg]
After AES: 41 63 89 36  62 ED D7 EB  CD 6E 15 C1  89 48 62 05
MIC tag  : 41 63 89 36  62 ED D7 EB  CD 6E
CTR Start: 01 00 00 00  0C 0B 0A 09  A0 A1 A2 A3  A4 A5 00 01
CTR[0001]: 0B 39 2B 9B  05 66 97 06  3F 12 56 8F  2B 13 A1 0F
CTR[0002]: 07 89 65 25  23 40 94 3B  9E 69 B2 56  CC 5E F7 31
CTR[MIC ]: 17 09 20 76  09 A0 4E 72  45 B3
Total packet length =   41. [Encrypted]
           00 01 02 03  04 05 06 07  08 09 0A 0B  07 34 25 94
           15 77 85 15  2B 07 40 98  33 0A BB 14  1B 94 7B 56
           6A A9 40 6B  4D 99 99 88  DD

=============== Packet Vector #11 ==================
AES Key:   C0 C1 C2 C3  C4 C5 C6 C7  C8 C9 CA CB  CC CD CE CF
      TA = A0 A1 A2 A3  A4 A5   48-bit pktNum = 0000.0D0C0B0A
Total packet length =   32. [Input (12 cleartext header octets)]
           00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
           10 11 12 13  14 15 16 17  18 19 1A 1B  1C 1D 1E 1F
CBC IV in: 61 00 00 00  0D 0C 0B 0A  A0 A1 A2 A3  A4 A5 00 14
CBC IV out:B0 84 85 79  51 D2 FA 42  76 EF 3A D7  14 B9 62 87
After xor: B0 88 85 78  53 D1 FE 47  70 E8 32 DE  1E B2 62 87   [hdr]
After AES: C9 B3 64 7E  D8 79 2A 5C  65 B7 CE CC  19 0A 97 0A
After xor: C5 BE 6A 71  C8 68 38 4F  71 A2 D8 DB  01 13 8D 11   [msg]
After AES: 34 0F 69 17  FA B9 19 D6  1D AC D0 35  36 D6 55 8B
After xor: 28 12 77 08  FA B9 19 D6  1D AC D0 35  36 D6 55 8B   [msg]
After AES: 6B 5E 24 34  12 CC C2 AD  6F 1B 11 C3  A1 A9 D8 BC
MIC tag  : 6B 5E 24 34  12 CC C2 AD  6F 1B
CTR Start: 01 00 00 00  0D 0C 0B 0A  A0 A1 A2 A3  A4 A5 00 01
CTR[0001]: 6B 66 BC 0C  90 A1 F1 12  FC BE 6F 4E  12 20 77 BC
CTR[0002]: 97 9E 57 2B  BE 65 8A E5  CC 20 11 83  2A 9A 9B 5B
CTR[MIC ]: 9E 64 86 DD  02 B6 49 C1  6D 37
Total packet length =   42. [Encrypted]
           00 01 02 03  04 05 06 07  08 09 0A 0B  67 6B B2 03
           80 B0 E3 01  E8 AB 79 59  0A 39 6D A7  8B 83 49 34
           F5 3A A2 E9  10 7A 8B 6C  02 2C

=============== Packet Vector #12 ==================
AES Key:   C0 C1 C2 C3  C4 C5 C6 C7  C8 C9 CA CB  CC CD CE CF
      TA = A0 A1 A2 A3  A4 A5   48-bit pktNum = 0000.0E0D0C0B
Total packet length =   33. [Input (12 cleartext header octets)]
           00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
           10 11 12 13  14 15 16 17  18 19 1A 1B  1C 1D 1E 1F
           20
CBC IV in: 61 00 00 00  0E 0D 0C 0B  A0 A1 A2 A3  A4 A5 00 15
CBC IV out:5F 8E 8D 02  AD 95 7C 5A  36 14 CF 63  40 16 97 4F
After xor: 5F 82 8D 03  AF 96 78 5F  30 13 C7 6A  4A 1D 97 4F   [hdr]
After AES: 63 FA BD 69  B9 55 65 FF  54 AA F4 60  88 7D EC 9F
After xor: 6F F7 B3 66  A9 44 77 EC  40 BF E2 77  90 64 F6 84   [msg]
After AES: 5A 76 5F 0B  93 CE 4F 6A  B4 1D 91 30  18 57 6A D7
After xor: 46 6B 41 14  B3 CE 4F 6A  B4 1D 91 30  18 57 6A D7   [msg]
After AES: 9D 66 92 41  01 08 D5 B6  A1 45 85 AC  AF 86 32 E8
MIC tag  : 9D 66 92 41  01 08 D5 B6  A1 45
CTR Start: 01 00 00 00  0E 0D 0C 0B  A0 A1 A2 A3  A4 A5 00 01
CTR[0001]: CC F2 AE D9  E0 4A C9 74  E6 58 55 B3  2B 94 30 BF
CTR[0002]: A2 CA AC 11  63 F4 07 E5  E5 F6 E3 B3  79 0F 79 F8
CTR[MIC ]: 50 7C 31 57  63 EF 78 D3  77 9E
Total packet length =   43. [Encrypted]
           00 01 02 03  04 05 06 07  08 09 0A 0B  C0 FF A0 D6
           F0 5B DB 67  F2 4D 43 A4  33 8D 2A A4  BE D7 B2 0E
           43 CD 1A A3  16 62 E7 AD  65 D6 DB
```

```
=============== Packet Vector #13 ==================
AES Key:   71 FB FD 78  FB E2 99 29  82 01 24 CC  71 44 75 7E
      TA = 42 EC 39 C1  86 99   48-bit pktNum = 2D38.1C9A0292
Total packet length =   31. [Input (8 cleartext header octets)]
           94 51 99 9F  03 E1 E7 2B  5F AE 94 A9  38 35 1C E8
           DF 8B E9 F5  D9 46 54 26  5A 67 74 8E  E6 31 F6
CBC IV in: 59 00 2D 38  1C 9A 02 92  42 EC 39 C1  86 99 00 17
CBC IV out:B0 E6 25 C9  37 B1 66 C5  70 79 3B 99  7D F0 C8 EC
After xor: B0 EE B1 98  AE 2E 65 24  97 52 3B 99  7D F0 C8 EC   [hdr]
After AES: 98 60 CE 17  C0 FE C7 9E  9B 00 8B 8A  99 BC 4C B2
After xor: C7 CE 5A BE  F8 CB DB 76  44 8B 62 7F  40 FA 18 94   [msg]
After AES: 42 5F 75 68  6D 69 31 EE  F6 B3 F4 3D  10 77 6F F4
After xor: 18 38 01 E6  8B 58 C7 EE  F6 B3 F4 3D  10 77 6F F4   [msg]
After AES: EF 93 3F 7F  9F B5 7D 54  BF 29 32 5A  3F 69 9C 5D
MIC tag  : EF 93 3F 7F  9F B5 7D 54
CTR Start: 01 00 2D 38  1C 9A 02 92  42 EC 39 C1  86 99 00 01
CTR[0001]: 9B 63 18 4C  23 A5 B1 18  49 71 1A 49  5C 40 DD DB
CTR[0002]: 2E F5 4D 53  86 73 A0 6E  A5 AD EB 84  D6 A9 37 02
CTR[MIC ]: 2F 45 06 56  3D 33 82 3B
Total packet length =   39. [Encrypted]
           94 51 99 9F  03 E1 E7 2B  C4 CD 8C E5  1B 90 AD F0
           96 FA F3 BC  85 06 89 FD  74 92 39 DD  60 42 56 C0
           D6 39 29 A2  86 FF 6F

=============== Packet Vector #14 ==================
AES Key:   71 FB FD 78  FB E2 99 29  82 01 24 CC  71 44 75 7E
      TA = 42 EC 39 C1  86 99   48-bit pktNum = 4DB9.0282DD86
Total packet length =   32. [Input (8 cleartext header octets)]
           50 D2 5E F4  B3 92 86 5A  06 F1 6B 83  83 88 72 91
           16 B6 F7 B8  4D 5D 44 1F  70 D6 8F 6B  A0 96 06 C3
CBC IV in: 59 00 4D B9  02 82 DD 86  42 EC 39 C1  86 99 00 18
CBC IV out:92 27 D3 5E  DD 64 94 B2  C9 6A 6F 0F  6F 3E AF DA
After xor: 92 2F 83 8C  83 90 27 20  4F 30 6F 0F  6F 3E AF DA   [hdr]
After AES: 9D 59 21 A7  EE 66 16 56  A6 4F D9 BA  5D 63 81 7A
After xor: 9B A8 4A 24  6D EE 64 C7  B0 F9 2E 02  10 3E C5 65   [msg]
After AES: 52 98 87 DB  DD 37 86 00  CE F4 83 C1  D1 8E 35 56
After xor: 22 4E 08 B0  7D A1 80 C3  CE F4 83 C1  D1 8E 35 56   [msg]
After AES: 46 AC 99 A0  50 35 91 70  1A A2 9E E0  B3 5F 72 9D
MIC tag  : 46 AC 99 A0  50 35 91 70
CTR Start: 01 00 4D B9  02 82 DD 86  42 EC 39 C1  86 99 00 01
CTR[0001]: 72 D0 3E 15  C3 F1 D5 65  66 32 A8 F2  CF A7 D1 9F
CTR[0002]: 52 69 9E 35  C9 C5 EE 07  70 80 67 C0  2B 38 41 20
CTR[MIC ]: E7 B7 A3 E1  84 B8 9C 6F
Total packet length =   40. [Encrypted]
           50 D2 5E F4  B3 92 86 5A  74 21 55 96  40 79 A7 F4
           70 84 5F 4A  82 FA 95 80  22 BF 11 5E  69 53 E8 C4
           A1 1B 3A 41  D4 8D 0D 1F

=============== Packet Vector #15 ==================
AES Key:   71 FB FD 78  FB E2 99 29  82 01 24 CC  71 44 75 7E
      TA = 42 EC 39 C1  86 99   48-bit pktNum = B5D4.B99983BA
Total packet length =   33. [Input (8 cleartext header octets)]
           6D 83 00 ED  50 09 A4 B2  6D E8 57 B7  58 49 19 CA
           EE 43 9C E4  8E BE 0C AC  00 F2 F9 32  50 0A 1C DD
           AC
CBC IV in: 59 00 B5 D4  B9 99 83 BA  42 EC 39 C1  86 99 00 19
CBC IV out:04 16 DE 1D  F7 77 E0 89  6E 07 B5 71  E9 1B 42 B2
After xor: 04 1E B3 9E  F7 9A B0 80  CA B5 B5 71  E9 1B 42 B2   [hdr]
After AES: 52 14 26 1E  6A 9D 50 38  D3 35 D5 76  0E ED E8 2E
After xor: 3F FC 71 A9  32 D4 49 F2  3D 76 49 92  80 53 E4 82   [msg]
After AES: 32 F2 0F FA  32 81 03 14  F9 CA FD C1  5E 37 27 0E
After xor: 32 00 F6 C8  62 8B 1F C9  55 CA FD C1  5E 37 27 0E   [msg]
After AES: 39 F5 F2 1E  2E 57 D7 14  96 46 57 CA  3B 70 A8 4C
MIC tag  : 39 F5 F2 1E  2E 57 D7 14
CTR Start: 01 00 B5 D4  B9 99 83 BA  42 EC 39 C1  86 99 00 01
CTR[0001]: 19 22 A3 83  B9 00 F2 DB  76 F3 84 65  D5 01 B4 C4
CTR[0002]: 50 6C 24 D4  0F 88 DB B0  68 98 12 E5  6E 64 A0 3B
```

```
CTR[MIC ]: 5B D9 B1 BB  D3 93 45 CA
Total packet length =   41. [Encrypted]
           6D 83 00 ED  50 09 A4 B2  74 CA F4 34  E1 49 EB 11
           98 B0 18 81  5B BF B8 68  50 9E DD E6  5F 82 C7 6D
           C4 62 2C 43  A5 FD C4 92  DE

=============== Packet Vector #16 ==================
AES Key:   71 FB FD 78  FB E2 99 29  82 01 24 CC  71 44 75 7E
      TA = 42 EC 39 C1  86 99   48-bit pktNum = AA65.BACC0941
Total packet length =   31. [Input (12 cleartext header octets)]
           EF 8F 46 B4  C9 77 98 32  BB F1 0A F1  C0 63 E7 C3
           DD 47 94 DF  53 A7 CD 68  CD 91 BF 29  04 4A 0B
CBC IV in: 59 00 AA 65  BA CC 09 41  42 EC 39 C1  86 99 00 13
CBC IV out:C5 95 2A 10  39 2B 60 9B  2C D5 30 83  CD 1D C8 FE
After xor: C5 99 C5 9F  7F 9F A9 EC  B4 E7 8B 72  C7 EC C8 FE   [hdr]
After AES: 41 D0 4D 56  FF DD D7 3D  AC CD AC 7D  63 64 3E 31
After xor: 81 B3 AA 95  22 9A 43 E2  FF 6A 61 15  AE F5 81 18   [msg]
After AES: 9C 86 E1 EE  BE 2B F0 BD  6D 11 20 3D  24 B1 B0 96
After xor: 98 CC EA EE  BE 2B F0 BD  6D 11 20 3D  24 B1 B0 96   [msg]
After AES: 3F 3A ED 74  AB C6 52 6A  DA C8 8D 14  0A 9F 84 23
MIC tag  : 3F 3A ED 74  AB C6 52 6A
CTR Start: 01 00 AA 65  BA CC 09 41  42 EC 39 C1  86 99 00 01
CTR[0001]: BD EF 70 9B  3C 70 A7 98  0F 36 C4 6E  7C D1 73 8D
CTR[0002]: 23 CC E5 E9  54 AD A2 09  21 17 FC 75  10 09 B3 E3
CTR[MIC ]: 38 17 B3 02  58 0A BA 84
Total packet length =   39. [Encrypted]
           EF 8F 46 B4  C9 77 98 32  BB F1 0A F1  7D 8C 97 58
           E1 37 33 47  5C 91 09 06  B1 40 CC A4  27 86 EE 07
           2D 5E 76 F3  CC E8 EE

=============== Packet Vector #17 ==================
AES Key:   71 FB FD 78  FB E2 99 29  82 01 24 CC  71 44 75 7E
      TA = 42 EC 39 C1  86 99   48-bit pktNum = F01B.307ADDDB
Total packet length =   32. [Input (12 cleartext header octets)]
           33 DF F8 40  E0 8C 16 9F  CB 1F F5 9F  B0 54 99 DD
           DD 6B EC 1E  13 2B 57 CB  0F DD 93 CD  E0 89 43 87
CBC IV in: 59 00 F0 1B  30 7A DD DB  42 EC 39 C1  86 99 00 14
CBC IV out:5C 16 AC 74  00 F3 24 1D  0F F1 5D 17  D2 CE 67 0E
After xor: 5C 1A 9F AB  F8 B3 C4 91  19 6E 96 08  27 51 67 0E   [hdr]
After AES: 8C 93 BC 6C  CA 8C 40 BB  03 FA 7C 0C  4F A0 10 42
After xor: 3C C7 25 B1  17 E7 AC A5  10 D1 2B C7  40 7D 83 8F   [msg]
After AES: 0C 03 5F 87  D7 DA 97 E5  77 7D D6 9C  EB 8C 84 86
After xor: EC 8A 1C 00  D7 DA 97 E5  77 7D D6 9C  EB 8C 84 86   [msg]
After AES: D0 8E 6D AC  0C 55 2B 34  F8 D3 05 82  B7 28 E5 C4
MIC tag  : D0 8E 6D AC  0C 55 2B 34
CTR Start: 01 00 F0 1B  30 7A DD DB  42 EC 39 C1  86 99 00 01
CTR[0001]: 3F 92 05 5E  E5 B1 2E F0  AF 6D C0 47  E8 FB 18 9E
CTR[0002]: C6 FD 0C C5  9F 93 37 F8  37 29 6A A6  E5 B7 00 F4
CTR[MIC ]: FD F5 FD 7C  00 82 8F 95
Total packet length =   40. [Encrypted]
           33 DF F8 40  E0 8C 16 9F  CB 1F F5 9F  8F C6 9C 83
           38 DA C2 EE  BC 46 97 8C  E7 26 8B 53  26 74 4F 42
           2D 7B 90 D0  0C D7 A4 A1

=============== Packet Vector #18 ==================
AES Key:   71 FB FD 78  FB E2 99 29  82 01 24 CC  71 44 75 7E
      TA = 42 EC 39 C1  86 99   48-bit pktNum = CE82.0B57FD4C
Total packet length =   33. [Input (12 cleartext header octets)]
           55 68 62 0F  19 A9 5D CB  98 4B C7 18  27 BF 59 E8
           8B FD 03 97  17 9F 7A CA  E6 B6 16 97  26 7A C0 5F
           24
CBC IV in: 59 00 CE 82  0B 57 FD 4C  42 EC 39 C1  86 99 00 15
CBC IV out:99 2D DF 68  2D 48 EF 2A  14 F0 16 6E  E4 14 9B 54
After xor: 99 21 8A 00  4F 47 F6 83  49 3B 8E 25  23 0C 9B 54   [hdr]
After AES: B7 97 9F B4  98 BC 07 E8  D2 60 92 00  1B 26 55 52
After xor: 90 28 C6 5C  13 41 04 7F  C5 FF E8 CA  FD 90 43 C5   [msg]
After AES: 2E 3E 5C 36  EA 3B B1 BA  0D 4F D0 EE  48 E7 38 DD
```

```
After xor: 08 44 9C 69  CE 3B B1 BA  0D 4F D0 EE  48 E7 38 DD   [msg]
After AES: 48 82 DE 1F  F0 3F 78 29  77 7C 01 A0  80 45 D1 D7
MIC tag  : 48 82 DE 1F  F0 3F 78 29
CTR Start: 01 00 CE 82  0B 57 FD 4C  42 EC 39 C1  86 99 00 01
CTR[0001]: 34 18 98 69  BD 1B AF 27  05 F2 7A C7  BF 2E F7 8A
CTR[0002]: 1E C6 81 EE  BC EE AF 2C  83 A1 37 C8  29 9B B1 DF
CTR[MIC ]: 62 C0 72 9E  52 D2 30 F3
Total packet length =   41. [Encrypted]
           55 68 62 0F  19 A9 5D CB  98 4B C7 18  13 A7 C1 81
           36 E6 AC B0  12 6D 00 0D  59 98 E1 1D  38 BC 41 B1
           98 2A 42 AC  81 A2 ED 48  DA

=============== Packet Vector #19 ==================
AES Key:   71 FB FD 78  FB E2 99 29  82 01 24 CC  71 44 75 7E
      TA = 42 EC 39 C1  86 99   48-bit pktNum = 34B5.2F55F836
Total packet length =   31. [Input (8 cleartext header octets)]
           30 27 70 18  36 DC FE E3  01 DE B7 F9  4D 49 E3 20
           BF AA C3 99  25 89 A5 6A  72 85 AE 03  CA 56 5D
CBC IV in: 61 00 34 B5  2F 55 F8 36  42 EC 39 C1  86 99 00 17
CBC IV out:07 03 FA 5A  50 F2 3C 36  E0 29 79 21  F4 B9 75 1B
After xor: 07 0B CA 7D  20 EA 0A EA  1E CA 79 21  F4 B9 75 1B   [hdr]
After AES: 2E 47 BB 82  95 84 25 CC  93 DD 77 9B  77 F2 D3 24
After xor: 2F 99 0C 7B  D8 CD C6 EC  2C 77 B4 02  52 7B 76 4E   [msg]
After AES: 3C 1D D1 EB  A5 E3 CB A0  14 93 CD C7  61 FC EB 29
After xor: 4E 98 7F E8  6F B5 96 A0  14 93 CD C7  61 FC EB 29   [msg]
After AES: F7 B0 EB A1  6C 26 4B 50  D4 DC 9F 6D  E1 B2 5B FE
MIC tag  : F7 B0 EB A1  6C 26 4B 50  D4 DC
CTR Start: 01 00 34 B5  2F 55 F8 36  42 EC 39 C1  86 99 00 01
CTR[0001]: 83 5D 2C BC  1E 6D A5 E8  BC 67 D3 56  33 F0 2B D1
CTR[0002]: E8 99 77 FC  10 10 49 92  3C FC 00 2A  85 79 A7 C0
CTR[MIC ]: 53 DD 0A 76  3B 12 C5 33  01 98
Total packet length =   41. [Encrypted]
           30 27 70 18  36 DC FE E3  82 83 9B 45  53 24 46 C8
           03 CD 10 CF  16 79 8E BB  9A 1C D9 FF  DA 46 14 A4
           6D E1 D7 57  34 8E 63 D5  44

=============== Packet Vector #20 ==================
AES Key:   71 FB FD 78  FB E2 99 29  82 01 24 CC  71 44 75 7E
      TA = 42 EC 39 C1  86 99   48-bit pktNum = 9BB8.8848BE25
Total packet length =   32. [Input (8 cleartext header octets)]
           54 FF D9 C2  A4 AE 72 B1  C9 33 92 50  20 D3 04 61
           5F B1 4A EF  9C 67 0E 0D  9F 8C D1 11  9D 25 69 5F
CBC IV in: 61 00 9B B8  88 48 BE 25  42 EC 39 C1  86 99 00 18
CBC IV out:EF 78 98 2E  7A 90 6E D4  72 A8 F4 11  8D E7 94 8A
After xor: EF 70 CC D1  A3 52 CA 7A  00 19 F4 11  8D E7 94 8A   [hdr]
After AES: D1 61 C3 62  F8 3C 51 3D  F3 FF 7F 1A  26 D4 F6 B9
After xor: 18 52 51 32  D8 EF 55 5C  AC 4E 35 F5  BA B3 F8 B4   [msg]
After AES: 46 CA 2F 4A  C4 99 EF C5  3B 5F FB 85  14 F7 BF 83
After xor: D9 46 FE 5B  59 BC 86 9A  3B 5F FB 85  14 F7 BF 83   [msg]
After AES: CD 55 F0 30  92 12 AE 02  EA 25 FA 94  87 DE 36 0F
MIC tag  : CD 55 F0 30  92 12 AE 02  EA 25
CTR Start: 01 00 9B B8  88 48 BE 25  42 EC 39 C1  86 99 00 01
CTR[0001]: E8 97 0A 1A  3A 73 B4 9F  89 E3 75 CB  F2 14 39 55
CTR[0002]: E9 CE 11 29  F6 5F 32 11  CD 7A 86 34  9C 67 F1 B5
CTR[MIC ]: 49 75 2B DA  6D 4A E9 9E  F8 4C
Total packet length =   42. [Encrypted]
           54 FF D9 C2  A4 AE 72 B1  21 A4 98 4A  1A A0 B0 FE
           D6 52 3F 24  6E 73 37 58  76 42 C0 38  6B 7A 5B 4E
           84 20 DB EA  FF 58 47 9C  12 69

=============== Packet Vector #21 ==================
AES Key:   71 FB FD 78  FB E2 99 29  82 01 24 CC  71 44 75 7E
      TA = 42 EC 39 C1  86 99   48-bit pktNum = 6E78.B6723686
Total packet length =   33. [Input (8 cleartext header octets)]
           AF BB B4 19  9C 13 D3 77  CE 25 C4 A7  B7 3B 06 1F
           58 6E 08 93  F9 17 8D CB  11 31 B2 E6  27 86 9A 4F
           44
```

```
CBC IV in: 61 00 6E 78  B6 72 36 86  42 EC 39 C1  86 99 00 19
CBC IV out:15 BF E5 B7  83 9A C6 00  B1 6F C9 F5  DA A8 3F 1C
After xor: 15 B7 4A 0C  37 83 5A 13  62 18 C9 F5  DA A8 3F 1C   [hdr]
After AES: E4 19 EF 1E  69 A1 48 EE  16 60 84 7D  D5 C9 D1 D6
After xor: 2A 3C 2B B9  DE 9A 4E F1  4E 0E 8C EE  2C DE 5C 1D   [msg]
After AES: 31 02 9A 8B  CA A3 07 1D  84 80 76 51  1D 9E 22 41
After xor: 20 33 28 6D  ED 25 9D 52  C0 80 76 51  1D 9E 22 41   [msg]
After AES: 21 5E E1 31  37 17 98 A5  FD 6E BB 74  D4 8E 59 C1
MIC tag  : 21 5E E1 31  37 17 98 A5  FD 6E
CTR Start: 01 00 6E 78  B6 72 36 86  42 EC 39 C1  86 99 00 01
CTR[0001]: 47 C1 8B 43  AF B6 3A C4  0A 7F CA C3  AE E4 83 0D
CTR[0002]: D9 91 74 F0  AE 23 37 4F  54 45 80 0D  27 0D A4 49
CTR[MIC ]: 17 E6 DC 69  6A 2E 09 B0  76 32
Total packet length =   43. [Encrypted]
           AF BB B4 19  9C 13 D3 77  89 E4 4F E4  18 8D 3C DB
           52 11 C2 50  57 F3 0E C6  C8 A0 C6 16  89 A5 AD 00
           10 36 B8 3D  58 5D 39 91  15 8B 5C

=============== Packet Vector #22 ==================
AES Key:   71 FB FD 78  FB E2 99 29  82 01 24 CC  71 44 75 7E
      TA = 42 EC 39 C1  86 99   48-bit pktNum = D079.0F8F3A99
Total packet length =   31. [Input (12 cleartext header octets)]
           5E C4 44 5A  EA D7 1B B1  DA 9E B5 10  0B 5E 8D 9F
           4F 27 49 CE  2B FC FF 25  B7 2C 81 17  55 CB 36
CBC IV in: 61 00 D0 79  0F 8F 3A 99  42 EC 39 C1  86 99 00 13
CBC IV out:47 4D BA 73  6A 5C 84 22  F5 0C 8B A3  60 72 F7 24
After xor: 47 41 E4 B7  2E 06 6E F5  EE BD 51 3D  D5 62 F7 24   [hdr]
After AES: A0 AF ED 92  55 EA 4C FC  5D 08 85 13  BE BF 07 25
After xor: AB F1 60 0D  1A CD 05 32  76 F4 7A 36  09 93 86 32   [msg]
After AES: F4 A9 E7 1A  D7 61 45 83  A0 CC 88 FA  25 5F B7 2D
After xor: A1 62 D1 1A  D7 61 45 83  A0 CC 88 FA  25 5F B7 2D   [msg]
After AES: 69 9C B6 66  03 78 1C 1B  92 93 86 F4  55 85 F4 6C
MIC tag  : 69 9C B6 66  03 78 1C 1B  92 93
CTR Start: 01 00 D0 79  0F 8F 3A 99  42 EC 39 C1  86 99 00 01
CTR[0001]: 30 4E B1 9A  EF 85 41 18  7E A7 77 F9  8D 0F BF E5
CTR[0002]: D1 8D 23 55  FA 2C 1C C7  F1 A5 86 A8  8E 7D 9E BF
CTR[MIC ]: 64 C3 13 58  1E EE F5 E8  E5 F2
Total packet length =   41. [Encrypted]
           5E C4 44 5A  EA D7 1B B1  DA 9E B5 10  3B 10 3C 05
           A0 A2 08 D6  55 5B 88 DC  3A 23 3E F2  84 46 15 0D
           5F A5 3E 1D  96 E9 F3 77  61

=============== Packet Vector #23 ==================
AES Key:   71 FB FD 78  FB E2 99 29  82 01 24 CC  71 44 75 7E
      TA = 42 EC 39 C1  86 99   48-bit pktNum = A625.D6288BF2
Total packet length =   32. [Input (12 cleartext header octets)]
           04 0C CF 5E  9E D7 4C EB  29 77 88 EB  E0 D2 59 4B
           F4 18 94 D9  BE 58 C4 EA  A3 BF 82 BF  A1 C5 3C 23
CBC IV in: 61 00 A6 25  D6 28 8B F2  42 EC 39 C1  86 99 00 14
CBC IV out:4F BC D9 D4  BB D2 77 FE  6B B3 CA 7A  AD 95 71 D2
After xor: 4F B0 DD D8  74 8C E9 29  27 58 E3 0D  25 7E 71 D2   [hdr]
After AES: E0 DD 09 D3  48 43 C1 70  E2 7C FE B0  4D 87 0A 66
After xor: 00 0F 50 98  BC 5B 55 A9  5C 24 3A 5A  EE 38 88 D9   [msg]
After AES: 4C 05 CA CC  DA 7D 5B 07  DE CA C7 14  D4 26 C5 D6
After xor: ED C0 F6 EF  DA 7D 5B 07  DE CA C7 14  D4 26 C5 D6   [msg]
After AES: 84 C8 29 3A  41 A2 E5 8C  6E 66 B2 26  BB B4 15 0D
MIC tag  : 84 C8 29 3A  41 A2 E5 8C  6E 66
CTR Start: 01 00 A6 25  D6 28 8B F2  42 EC 39 C1  86 99 00 01
CTR[0001]: 12 62 98 95  AB 26 D2 51  4C 32 59 F4  8E 21 19 4C
CTR[0002]: 1E 70 D2 AF  FE 0A 84 D5  45 27 C4 25  75 5B 99 5D
CTR[MIC ]: 0F EE 6A 8E  47 D2 BC 52  C9 CB
Total packet length =   42. [Encrypted]
           04 0C CF 5E  9E D7 4C EB  29 77 88 EB  F2 B0 C1 DE
           5F 3E 46 88  F2 6A 9D 1E  2D 9E 9B F3  BF B5 EE 8C
           8B 26 43 B4  06 70 59 DE  A7 AD

=============== Packet Vector #24 ==================
```

```
AES Key:   71 FB FD 78  FB E2 99 29  82 01 24 CC  71 44 75 7E
      TA = 42 EC 39 C1  86 99   48-bit pktNum = 7CD9.622F4AED
Total packet length =   33. [Input (12 cleartext header octets)]
           AB CB 0A 7E  49 E6 F8 74  E7 1D AA 1A  CC 96 CA 13
           39 66 05 81  59 70 D4 65  1D 28 88 00  F2 35 DA 22
           63
CBC IV in: 61 00 7C D9  62 2F 4A ED  42 EC 39 C1  86 99 00 15
CBC IV out:6B 58 00 94  F8 F3 99 9C  9E 23 D0 58  57 E8 F9 58
After xor: 6B 54 AB 5F  F2 8D D0 7A  66 57 37 45  FD F2 F9 58   [hdr]
After AES: 75 6A 35 9E  7F 06 79 D1  16 9E 8B FF  A1 4B 7C F1
After xor: B9 FC FF 8D  46 60 7C 50  4F EE 5F 9A  BC 63 F4 F1   [msg]
After AES: 00 12 C1 1D  3B 6F D0 B5  8E 72 2F A4  DB 2A 91 29
After xor: F2 27 1B 3F  58 6F D0 B5  8E 72 2F A4  DB 2A 91 29   [msg]
After AES: 65 71 83 09  48 3B 45 14  9C 05 90 A9  C7 96 56 E4
MIC tag  : 65 71 83 09  48 3B 45 14  9C 05
CTR Start: 01 00 7C D9  62 2F 4A ED  42 EC 39 C1  86 99 00 01
CTR[0001]: A7 50 18 88  92 3F 63 B0  DA ED 59 36  2D 61 93 50
CTR[0002]: 8C 32 57 34  AB 75 8E AB  57 A7 DB B0  F2 41 EA AD
CTR[MIC ]: D8 39 8F F8  7A 1C 3F 34  E5 94
Total packet length =   43. [Encrypted]
           AB CB 0A 7E  49 E6 F8 74  E7 1D AA 1A  6B C6 D2 9B
           AB 59 66 31  83 9D 8D 53  30 49 1B 50  7E 07 8D 16
           C8 BD 48 0C  F1 32 27 7A  20 79 91
```

F.8. Suggested pass-phrase-to-preshared-key mapping

F.8.1 Introduction

The RSN pre-shared key consists of 256 bits, or 64 bytes when represented in hex. It is difficult for a user to correctly enter 64 hex characters. Most users, however, are familiar with passwords and pass-phrases, and feel more comfortable entering them than entering keys. A user is more likely to be able to enter an ASCII password or pass-phrase, even though doing so limits the set of possible keys. This suggests that the best that can be done is to introduce a pass-phrase to preshared key mapping.

This clause defines a pass-phrase to preshared key mapping that is the preferred mechanism of this sort for RSN and TSN networks. This pass-phrase mapping was introduced to encourage users unfamiliar with cryptographic concepts to enable the security features of their WLAN.

A pass-phrase typically has about 2.5 bits of security per character, so the pass-phrase mapping converts an n byte password into a key with about $2.5n + 12$ bits of security. Hence, it provides a relatively low level of security, with keys generated from short passwords subject to dictionary attack. Use of the key hash is recommended only for IT-less environments. A key generated from a pass-phrase of less than about 20 characters is unlikely to deter attacks against small businesses and enterprises.

The pass-phrase mapping defined here uses the PBKDF2 method from PKCS #5 v2.0: Password-based Cryptography Standard.

$$PSK = \mathrm{PBKDF2}(PassPhrase, ssid, ssidLength, 4096, 256)$$

- PassPhrase is an ASCII string which has a minimum of 8 and a maximum of 63 characters not including the null terminator. The limit of 63 characters comes from the fact that 256 bits is represented by 64 characters in hex.

  PassPhrase should consist of characters from the following three groups

<table>
<tr><th>Group</th><th>Examples</th></tr>
<tr><td>Letters (upper and lower case)</td><td>A, B, C, … (and a, b, c,…)</td></tr>
<tr><td>Numerals</td><td>0, 1, 2, 3, 4, 5, 6, 7, 8, 9</td></tr>
<tr><td>Symbols (all characters not defined as letters or numerals)</td><td>`~!@#$%^&*()_+=-{}|[]\”:;’<>?,./</td></tr>
</table>

- *ssid* is the SSID of ESS or IBSS where this pass-phrase is in use, encoded as the hex string used in the Beacons and Probe Responses for the ESS or IBSS. Implementations of this pass-phrase mapping should by default define a unique SSID for each AP, e.g., the BSSID of the AP, and non-AP STA’s should learn the SSID from the AP, so the user need not enter it.
- *ssidlength* is the number of octets of the *ssid*.
- 4096 is the number of times the pass-phrase is hashed.
- 256 is the number of bits output by the pass-phrase mapping.

F.8.2 Reference implementation

```
/*
 * F(P, S, c, i) = U1 xor U2 xor ... Uc
 * U1 = PRF(P, S || Int(i))
 * U2 = PRF(P, U1)
 * Uc = PRF(P, Uc-1)
 */

void F(
	char *password,
	unsigned char *ssid,
	int ssidlength,
	int iterations,
	int count,
	unsigned char *output)
{
	unsigned char digest[36], digest1[A_SHA_DIGEST_LEN];
	int i, j;

	/* U1 = PRF(P, S || int(i)) */
	memcpy(digest, ssid, ssidlength);
	digest[ssidlength] = (unsigned char)((count>>24) & 0xff);
	digest[ssidlength+1] = (unsigned char)((count>>16) & 0xff);
	digest[ssidlength+2] = (unsigned char)((count>>8) & 0xff);
	digest[ssidlength+3] = (unsigned char)(count & 0xff);
	hmac_sha1((unsigned char*) password, (int) strlen(password),
	    digest, ssidlength+4, digest1);

	/* output = U1 */
	memcpy(output, digest1, A_SHA_DIGEST_LEN);

	for (i = 1; i < iterations; i++) {
		/* Un = PRF(P, Un-1) */
		hmac_sha1((unsigned char*) password, (int) strlen(password),
		    digest1, A_SHA_DIGEST_LEN, digest);
		memcpy(digest1, digest, A_SHA_DIGEST_LEN);
```

```
                /* output = output xor Un */
                for (j = 0; j < A_SHA_DIGEST_LEN; j++) {
                        output[j] ^= digest[j];
                }
        }
}

/*
 * password - ascii string up to 63 characters in length
 * ssid - octet string up to 32 octets
 * ssidlength - length of ssid in octets
 * output must be 40 octets in length and outputs 256 bits of key
 */
int PasswordHash (
        char *password,
        unsigned char *ssid,
        int ssidlength,
        unsigned char *output)
{
        if ((strlen(password) > 63) || (ssidlength > 32))
                return 0;

        F(password, ssid, ssidlength, 4096, 1, output);
        F(password, ssid, ssidlength, 4096, 2,
             &output[A_SHA_DIGEST_LEN]);
        return 1;
}
```

F.8.3 Test vectors

Test case 1
Pass Phrase = “password”
SSID = { ‘I’, ‘E’, ‘E’ ‘E’ }
SSIDLength = 4
PSK =
`534036bd932a231c80f8b52ccb18ce0d17cc78fc4675c7b4dfa4396540111450`

Test case 2
Pass Phrase = “ThisIsAPassword”
SSID = { ‘T’, ‘h’, ‘i’, ‘s’, ‘I’, ‘s’, ‘A’, ‘S’, ‘S’, ‘I’, ‘D’ }
SSIDLength = 11
PKS =
`520f0426ee757e8dfbb254e17971409a66969b2483f7492b5342dcce682b1155`

Test case 3
Password = “aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa”
SSID = {‘Z’,‘Z’,‘Z’,‘Z’, ‘Z’,‘Z’,‘Z’,‘Z’, ‘Z’,‘Z’,‘Z’,‘Z’, ‘Z’,‘Z’,‘Z’,‘Z’, ‘Z’,‘Z’,‘Z’,‘Z’, ‘Z’,‘Z’,‘Z’,‘Z’, ‘Z’,‘Z’,‘Z’,‘Z’,‘Z’,‘Z’,‘Z’,‘Z’}
SSIDLength = 32
PKS =
`b4266b172c373a47260ee97faa0d199aaba2a31dbe5fc5a8becc1784857c0fbc`

F.9. Suggestions for random number generation

In order to properly implement cryptographic protocols, every platform needs the ability to generate cryptographic quality random numbers. RFC 1750 explains the notion of cryptographic quality random numbers and provides advice on ways to harvest suitable randomness. It recommends sampling multiple sources each of which contains some randomness, and by passing the complete set of samples through a pseudo-random function. By following this advice, an implementation can usually collect enough randomness to distill into a seed for a pseudo-random number generator whose output will be unpredictable.

This annex suggests two sample techniques that can be combined with the other recommendations of RFC 1750 to harvest randomness. The first method is a software solution that can be implemented on most hardware; the second is a hardware-assisted solution. These solutions are expository only, to demonstrate that it is feasible to harvest randomness on any 802.11 platform. They are not mutually exclusive, and they do not preclude the use of other sources of randomness when available; in this case, the more the merrier. As many sources of randomness as possible should be gathered into a buffer, and then hashed, to obtain a seed for the pseudo-random number generator.

F.9.1 Software Sampling

Due to the nature of clock circuits in modern electronics, there will be some lack of correlation between two clocks in two different pieces of equipment, even when high quality crystals are used—crystal clocks are subject to jitter, noise, drift, and frequency mismatch. This randomness may be as little as the placement of the clock waveform edges. Even if one entity were to attempt to synchronize itself to another entity's clock, the correlation cannot be perfect, due to noise and uncertainties of the synchronization.

Two clock circuits in the same piece of equipment may synchronize in frequency, but again the correlation will not be perfect due to the noise and jitter of the circuits.

The randomness between the two clocks may not be much per sample—a tenth of a bit or less—but enough samples may be collected to gather enough randomness to form a seed.

A device can use software methods to take advantage of this lack of synchronization, to collect randomness from different sources. As an example, an AP might measure the packet arrival times on a Ethernet wireless ports. There is always some amount of traffic on modern Ethernets: ARPs, DHCP requests, NetBIOS advertisements, etc. The following example algorithm takes this traffic. In the example, an AP obtains randomness from the available traffic; if Ethernet traffic is available, the AP measures that for randomness; otherwise it waits for the first association and creates traffic that it can obtain randomness from.

The clocks used to time the packets should be the highest resolution available, preferable 1ms resolution or better. The clock used to time packet arrival should not be related to the clock used for packet serialization.

```
Initialize result to empty array
LoopCounter = 0
Wait until Ethernet traffic or association
Repeat until global key counter "random enough" or 32 times {
      result = PRF-256(0, "Init Counter",
            Local Mac Address || Time || result || LoopCounter)
      LoopCounter++
      Repeat 32 times {
            If Ethernet traffic available then
                  Take lowest byte of time when Ethernet packet is seen
                  Concatenate the seen time onto result
            else
                  Start 4-way handshake, aborting after message 2
                  Take lowest byte of time of when message 1 is sent
                  Take lowest byte of time of when message 2 is
                        received
                  Take lowest byte of RSSI from message 2
                  Take SNonce from message 2
                  Concatenate the sent time; receive time, RSSI and
                        SNonce onto result
      }
}
Global key counter = result = PRF-256(0, "Init Counter",
      Local Mac Address || Time || result || LoopCounter)
```

Note: The Time may be 0 if it is not available.

F.9.2 Hardware Assisted Solution

This example implementation uses hardware ring oscillators to generate randomness, as depicted in below.

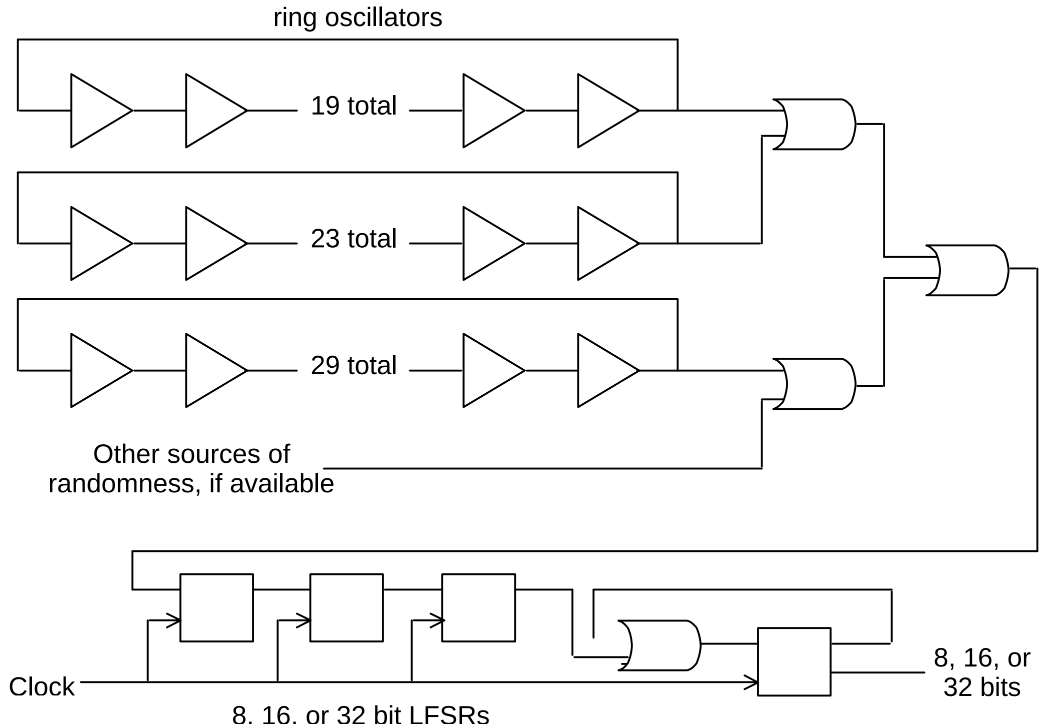


Figure 55—Randomness generating circuit

The above circuit generates randomness. The clock input should be about the same frequency as the ring oscillator's natural frequencies. The LFSR should be chosen to be one that is maximal length. Example LFSRs can be found at http://www-2.cs.cmu.edu/~koopman/lfsr/.

The three ring oscillators should be isolated from each other as much as possible, to avoid harmonic locking between them. In addition, the three ring oscillators should not be near any other clock circuitry within the system, to avoid these ring oscillators locking to system clocks.

The output of the LFSR is read by software and concatenated until enough randomness is collected. As a rule of thumb, reading from the LFSR eight to sixteen times the number of bits as the desired number of random bits is sufficient.

```
Initialize result to empty array
Repeat 1024 times {
        Read LFSR
        result = result | LFSR
        Wait a time period
}
Global key counter = PRF-256(0, "Init Counter", result)
```

F.10. Additional test vectors

F.10.1 Notation

In the examples here, frames are represented as a stream of octets, each octet in hex notation, sometimes with text annotation. The order of transmission for octets is left to right, top to bottom. For example, consider the following representation of a frame:

Description #1	00 01 02 03
	04 05
Description #2	06 07 08

The frame consists of nine octets, represented in hex notation as “00”, “01”, ..., “08”. The octet represented by “00” is transmitted first, and the octet represented by “08” is transmitted last. Similar tables are used for other purposes, such as describing a cryptographic operation.

In the text discussion outside of tables, integer values are represented in either hex notation using an “0x” prefix or in decimal notation using no prefix. For example, the hex notation 0x12345 and the decimal notation 74565 represent the same integer value.

F.10.2 WEP Encapsulation

The discussion here represents an RC4 encryption using a table that shows the key, plaintext input, and ciphertext output. For reference, here is a table that describes test vector “Commerce” of <draft-kaukonen-cipher-arcfour-03.txt>, a work-in-progress.

Key	61 8a 63 d2 fb
Plaintext	dc ee 4c f9 2c
Ciphertext	f1 38 29 c9 de

The MPDU data, prior to WEP encapsulation, is as follows:

MPDU data	aa aa 03 00 00 00 08 00 45 00 00 4e 66 1a 00 00 80 11 be 64 0a 00 01 22 0a ff ff ff 00 89 00 89 00 3a 00 00 80 a6 01 10 00 01 00 00 00 00 00 00 20 45 43 45 4a 45 48 45 43 46 43 45 50 46 45 45 49 45 46 46 43 43 41 43 41 43 41 43 41 43 41 41 41 00 00 20 00 01

RC4 encryption is performed as follows:

Key	fb 02 9e 30 31 32 33 34
Plaintext	aa aa 03 00 00 00 08 00 45 00 00 4e 66 1a 00 00 80 11 be 64 0a 00 01 22 0a ff ff ff 00 89 00 89 00 3a 00 00 80 a6 01 10 00 01 00 00 00 00 00 00 20 45 43 45 4a 45 48 45 43 46 43 45 50 46 45 45 49 45 46 46 43 43 41 43 41 43 41 43 41 43 41 41 41 00 00 20 00 01 1b d0 b6 04
Ciphertext	f6 9c 58 06 bd 6c e8 46 26 bc be fb 94 74 65 0a ad 1f 79 09 b0 f6 4d 5f 58 a5 03 a2 58 b7 ed 22 eb 0e a6 49 30 d3 a0 56 a5 57 42 fc ce 14 1d 48 5f 8a a8 36 de a1 8d f4 2c 53 80 80 5a d0 c6 1a 5d 6f 58 f4 10 40 b2 4b 7d 1a 69 38 56 ed 0d 43 98 e7 ae e3 bf 0e 2a 2c a8 f7

The plaintext consists of the MPDU data, followed by a 4-octet CRC-32 calculated over the MPDU data.

The expanded MPDU, after WEP encapsulation, is as follows:

IV	fb 02 9e 80
MPDU data	f6 9c 58 06 bd 6c e8 46 26 bc be fb 94 74 65 0a ad 1f 79 09 b0 f6 4d 5f 58 a5 03 a2 58 b7 ed 22 eb 0e a6 49 30 d3 a0 56 a5 57 42 fc ce 14 1d 48 5f 8a a8 36 de a1 8d f4 2c 53 80 80 5a d0 c6 1a 5d 6f 58 f4 10 40 b2 4b 7d 1a 69 38 56 ed 0d 43 98 e7 ae e3 bf 0e
ICV	2a 2c a8 f7

The IV consists of the first three octets of the RC4 key, followed by an octet containing the KeyID value in the upper two bits. In this example, the KeyID value is 2. The MPDU data consists of the ciphertext, excluding the last four octets. The ICV consists of the last four octets of the ciphertext, which is the encrypted CRC-32 value.

F.10.3 TKIP encapsulation

The discussion here represents a Michael calculation using a table that shows the key, input data, and MIC output. For reference, here is a table that describes a the test vector for input string "Michael" shown in Annex F:

Key	d5 5e 10 05 10 12 89 86
Input data	4d 69 63 68 61 65 6c
MIC	0a 94 2b 12 4e ca a5 46

The discussion represents calculation of phase 2 of the temporal key mixing function using a table that shows the TTAK key, the IV input, and the RC4-key output. For reference, here is a table that describes test vector #4 shown in Annex F:

TTAK	a2 db 10 2a 3e a3 56 82 99 56 c4 5d 7b 11 fc 54
IV	55 c6
RC4 key	55 75 c6 a5 04 2b 11 29 25 1e 22 f4 5a 25 c7 d6

The MSDU data, prior to TKIP encapsulation, is as follows:

MSDU data	aa aa 03 00 00 00 08 00 45 00 00 4e 66 1a 00 00 80 11 be 64 0a 00 01 22 0a ff ff ff 00 89 00 89 00 3a 00 00 80 a6 01 10 00 01 00 00 00 00 00 00 20 45 43 45 4a 45 48 45 43 46 43 45 50 46 45 45 49 45 46 46 43 43 41 43 41 43 41 43 41 43 41 41 41 00 00 20 00 01

The MIC is computed using Michael, as follows:

Key	d5 5e 10 05 10 12 89 86
Input data	aa aa 03 00 00 00 08 00 45 00 00 4e 66 1a 00 00 80 11 be 64 0a 00 01 22 0a ff ff ff 00 89 00 89 00 3a 00 00 80 a6 01 10 00 01 00 00 00 00 00 00 20 45 43 45 4a 45 48 45 43 46 43 45 50 46 45 45 49 45 46 46 43 43 41 43 41 43 41 43 41 43 41 41 41 00 00 20 00 01
MIC	31 2d 0f fb 8c d6 58 30

The input to the MIC calculation is the MSDU data.

The MSDU and MIC are concatenated, and if necessary, the concatenated result is fragmented into several MPDUs. In this example, it is fragmented into two MPDUs, as follows:

MPDU #1 data	aa aa 03 00 00 00 08 00 45 00 00 4e 66 1a 00 00 80 11 be 64 0a 00 01 22 0a ff ff ff 00 89 00 89 00 3a 00 00 80 a6 01 10 00 01 00 00 00 00 00

MPDU #2 data	00 20 45 43 45 4a 45 48 45 43 46 43 45 50 46 45 45 49 45 46 46 43 43 41 43 41 43 41 43 41 43 41 41 41 00 00 20 00 01 31 2d 0f fb 8c d6 58 30

To encrypt the first MPDU, the RC4 key is derived using phase 2 of the temporal key mixing function, as follows:

TTAK	a2 db 10 2a 3e a3 56 82 99 56 c4 5d 7b 11 fc 54
IV	5b a0
RC4 key	5b 7b a0 d7 9a ee c2 2e 0d d1 a9 14 bd b8 42 30

In this example, the IV has value 23456, or 0x5ba0.

RC4 encryption of the first MPDU is performed as follows:

Key	5b 7b a0 d7 9a ee c2 2e 0d d1 a9 14 bd b8 42 30
Plaintext	aa aa 03 00 00 00 08 00 45 00 00 4e 66 1a 00 00 80 11 be 64 0a 00 01 22 0a ff ff ff 00 89 00 89 00 3a 00 00 80 a6 01 10 00 01 00 00 00 00 00 99 22 5f 4e

Ciphertext	e0 3f 0e 76 ce dd d5 54 cb 7d af 74 41 8f 9f db 86 ed 6a 46 f1 1c e0 6a 64 53 3e 95 76 43 3a 93 ac e5 5d 65 ac f0 8e ec 87 88 e7 a8 ad f6 04 ee 4b 64 6e

The plaintext consists of the MPDU data, followed by a 4-octet CRC-32 calculated over the MPDU data.

The expanded first MPDU, after encapsulation, is as follows:

IV	5b 7b a0 40
MPDU #1 data	e0 3f 0e 76 ce dd d5 54 cb 7d af 74 41 8f 9f db 86 ed 6a 46 f1 1c e0 6a 64 53 3e 95 76 43 3a 93 ac e5 5d 65 ac f0 8e ec 87 88 e7 a8 ad f6 04
ICV	ee 4b 64 6e

The IV field consists of the first three octets of the RC4 key, followed by an octet containing the KeyID field in the upper two bits. In this example, the KeyID has value 1. The MPDU data consists of the ciphertext, excluding the last four octets. The ICV consists of the last four octets of the ciphertext, which is the encrypted CRC-32 value.

To encrypt the second MPDU, the RC4 key is derived using phase 2 of the temporal key mixing function, as follows:

TTAK	a2 db 10 2a 3e a3 56 82 99 56 c4 5d 7b 11 fc 54
IV	5b a1
RC4 key	5b 7b a1 2c 67 9b cb 70 e7 c3 d6 5e 14 d5 2a c7

The IV for the second MPDU is the value of the IV for the first MPDU, plus one.

RC4 encryption of the second MPDU is performed in the same manner as for the first MPDU, as follows:

Key	5b 7b a1 2c 67 9b cb 70 e7 c3 d6 5e 14 d5 2a c7
Plaintext	00 20 45 43 45 4a 45 48 45 43 46 43 45 50 46 45 45 49 45 46 46 43 43 41 43 41 43 41 43 41 43 41 41 41 00 00 20 00 01 31 2d 0f fb 8c d6 58 30 b6 d3 a7 06
Ciphertext	9f 26 25 79 b8 bf 49 9e 27 bc a6 a9 2c 4d 21 95 4b 3b 84 45 c0 77 33 11 f1 78 ff 14 57 83 15 3c a0 93 31 81 ac 2d bb 1c 81 cc 0e 0b e3 60 06 04 98 9c dc

The expanded second MPDU, after encapsulation, is similar to that of the first MPDU, as follows:

IV	5b 7b a1 40
MPDU #2 data	9f 26 25 79 b8 bf 49 9e 27 bc a6 a9 2c 4d 21 95 4b 3b 84 45 c0 77 33 11 f1 78 ff 14 57 83 15 3c a0 93 31 81 ac 2d bb 1c 81 cc 0e 0b e3 60 06
ICV	04 98 9c dc

F.10.4 AES-CCMP

F.10.4.1 AES-CCMP Encapsulation Example

The MPDU parameters and data, prior to AES-CCM encapsulation, is as follows:

```
Type     = 2   SubType  = 11
ToDS     = 1   FromDS   = 1
MoreFrag = 1   Retry    = 1
PwrMgt   = 1   moreData = 1
WEP      = 1
Order    = 1
Duration = 200
A1 = a1:a1:a1:a1:a1:a1
A2 = a2:a2:a2:a2:a2:a2
A3 = a3:a3:a3:a3:a3:a3
seqNum = 4000  fraqNum = 1
A4 = a4:a4:a4:a4:a4:a4
QC = 0xff77
```

```
QoS-TID  = 7  QoS-FEC = 1
QoS-AckP = 3  QoS-TXOP/QL = 0xff
Algorithm = AES_CCM
KeyId = 1
PN    = 0x000000000001  (decimal = 1 )
Data  =
    69 6e 6f 76 61 74 69 6f  6e 73 20 69 6e 20 77 69
    72 65 6c 65 73 73
```

The calculation of the encrypted MPDU is as follows:

CCM additional auth data (muted header)	b8 c7 a1 a1 a1 a1 a1 a1 a2 a2 a2 a2 a2 a2 a3 a3 a3 a3 a3 a3 01 fa a4 a4 a4 a4 a4 a4 07 00
CCM Nonce Value	07 a2 a2 a2 a2 a2 a2 00 00 00 00 00 01
Encryption Header Note PN is big-endian!!	00 00 00 60 00 00 00 01
CBC Input Blocks	59 07 a2 a2 a2 a2 a2 a2 00 00 00 00 00 01 00 16 00 1e b8 c7 a1 a1 a1 a1 a1 a1 a2 a2 a2 a2 a2 a2 a3 a3 a3 a3 a3 a3 01 fa a4 a4 a4 a4 a4 a4 07 00 69 6e 6f 76 61 74 69 6f 6e 73 20 69 6e 20 77 69 72 65 6c 65 73 73 00 00 00 00 00 00 00 00 00 00
CBC MIC Value	a6 de 98 74 73 da 55 34 5b f0 26 e6 f0 b8 d9 27
CTR Mode Preload (0)	01 07 a2 a2 a2 a2 a2 a2 00 00 00 00 00 01 00 00
CCM Final MIC Value	7d 63 5f d0 d8 3f 8b 6c
CTR Mode Data to Encrypt	69 6e 6f 76 61 74 69 6f 6e 73 20 69 6e 20 77 69 72 65 6c 65 73 73
CTR Mode Encrypted Data	6b 64 99 64 53 85 64 f1 28 69 08 ab fb 12 41 ed 10 04 d4 44 da 3f
CCM Encrypted MPDU with FCS	b8 ff c8 00 a1 a1 a1 a1 a1 a1 a2 a2 a2 a2 a2 a2 a3 a3 a3 a3 a3 a3 01 fa a4 a4 a4 a4 a4 a4 77 ff 00 00 00 60 00 00 00 01 6b 64 99 64 53 85 64 f1 28 69 08 ab fb 12 41 ed 10 04 d4 44 da 3f 7d 63 5f d0 d8 3f 8b 6c d5 67 81 13

F.10.4.2 Additional CCMP Vest Vectors

The following CCMP test vectors are full 802.11 CCMP encrypted MPDUs. The MPDUs and CCMP processing can be tested by using the supplied key to decrypt the MPDU and checking the MIC value.

Description	ccm#0001 : Data Packet, no A4 and no QC
Key	C0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf
CCMP Encrypted MPDU with FCS	08 41 02 01 00 06 25 a7 c4 36 00 02 2d 49 97 b4 00 06 25 a7 c4 36 e0 00 06 05 00 a0 04 03 02 01 1e e5 2d 13 b1 be 3f 20 42 5b 3f de dd d4 55 2b 98 71 d8 7b 65 8c fd 57 f7 96 ad 71 87

Description	ccm#0002 : Data Packet, no A4 and no QC, retry
Key	c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf
CCMP Encrypted MPDU with FCS	08 49 02 01 00 06 25 a7 c4 36 00 02 2d 49 97 b4 00 06 25 a7 c4 36 e0 00 06 05 00 e0 04 03 02 01 1e e5 2d 13 b1 be 3f 20 42 5b 3f de dd d4 55 2b 98 71 d8 7b 65 8c fd 57 f7 67 c5 18 73

Description	ccm#0003 : Data Packet,A4 with no QC
Key	c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf
CCMP Encrypted MPDU with FCS	08 43 02 01 00 06 25 a7 c4 36 00 02 2d 49 97 b4 00 06 25 a7 c4 36 e0 00 41 42 43 44 45 46 00 00 00 20 00 00 00 01 3b e9 b2 46 c6 fc 7a 51 55 1e 14 c6 a8 85 28 bc 06 56 67 c8 ef 30 b3 12 69 14 6c 3b c3

Description	ccm#0004 : Data Packet,A4 and QC
Key	c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf
CCMP Encrypted MPDU with FCS	88 43 02 01 00 06 25 a7 c4 36 00 02 2d 49 97 b4 00 06 25 a7 c4 36 e0 00 41 42 43 44 45 46 04 00 00 00 00 20 00 00 00 01 46 72 f2 9e 41 54 e9 11 58 47 c2 a9 ae dc 10 0c e8 82 53 bd a2 04 ae 1d 33 05 af 02 1e

Description	ccm#0005 : Data Packet,QC no A4
Key	c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf
CCMP Encrypted MPDU with FCS	88 41 02 01 00 06 25 a7 c4 36 00 02 2d 49 97 b4 00 06 25 a7 c4 36 e0 00 04 00 00 00 00 20 00 00 00 01 46 72 f2 9e 41 54 e9 11 58 47 c2 a9 ae dc 10 0c e8 dc 91 98 bf 6a 52 c8 03 67 12 0b 83

Description	ccm#0006 : Data Packet, no A4, No QC, look out for the C9
Key	00 01 02 03 04 05 06 07 08 **c9** 0a 0b 0c 0d 0e 0f
CCMP Encrypted MPDU with FCS	08 41 02 01 00 06 25 a7 c4 36 00 02 2d 49 97 b4 00 06 25 a7 c4 36 e0 00 06 05 00 a0 04 03 02 01 de bf 2c c9 94 e6 5a 70 2c ee e3 19 84 21 39 c3 f2 9a 2e 12 63 11 74 5f 3c 20 3d fd 4e

Description	ccm#0007 : Data Packet, same as 144r4 data, odd a4 alignment
Key	00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
CCMP Encrypted MPDU with FCS	08 43 12 34 ff ff ff ff ff ff 00 40 96 45 07 f1 08 00 46 17 62 3e 50 67 aa aa 03 00 00 00 00 05 00 a0 04 03 02 01 22 3b 8c 39 95 b0 d0 c5 81 d7 19 2f e4 4a ad 02 76 61 30 fe 1a 2c 1d 54 0b e2 ce 2f 4d 53 03 1b 62 68 8f 9d 75 81 08 ff 6d 35 e5 a0 75 f4 c2 0a 95 d2 f2 c7 45 94 b6 9e 64 63 3a fa 6e 5c 97 57 ea 49 24 66 f4 e5 3e e9 81 77 d2 0b f9 d9 82 15 ac ce 8f e8 7b 7e f1 ef ae cc 9b ac

Description	ccm#0008 : All flag bits set with QC
Key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
CCMP Encrypted MPDU with FCS	b8 ff c8 00 a1 a1 a1 a1 a1 a1 a2 a2 a2 a2 a2 a2 a3 a3 a3 a3 a3 a3 01 fa a4 a4 a4 a4 a4 a4 77 ff 00 00 00 60 00 00 00 01 6b 64 99 64 53 85 64 f1 28 69 08 ab fb 12 41 ed 10 04 d4 44 da 3f 7d 63 5f d0 d8 3f 8b 6c d5 67 81 13

F.10.5 AES-OCB encapsulation

The discussion here represents an AES-OCB encryption using a table that shows the key, nonce input, plaintext input, ciphertext output, and tag output. For reference, here is a table that describes test case “OCB-AES-128-34B” available at `http://www.cs.ucdavis.edu/~rogaway/ocb/`.

Key	00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
Nonce	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01
Plaintext	00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20 21
Ciphertext	01 a0 75 f0 d8 15 b1 a4 e9 c8 81 a1 bc ff c3 eb d4 90 3d d0 02 5b a4 aa 83 7c 74 f1 21 b0 26 0f a9 5d
Tag	cf 83 41 bb 10 82 0c cf 14 bd ec 56 b8 d7 d6 ab

The MSDU data, prior to AES-OCB encapsulation, is as follows:

MSDU data	aa aa 03 00 00 00 08 00 45 00 00 4e 66 1a 00 00 80 11 be 64 0a 00 01 22 0a ff ff ff 00 89 00 89 00 3a 00 00 80 a6 01 10 00 01 00 00 00 00 00 00 20 45 43 45 4a 45 48 45 43 46 43 45 50 46 45 45 49 45 46 46 43 43 41 43 41 43 41 43 41 43 41 41 41 00 00 20 00 01

In this example, the following parameters will be used:

- Replay counter value is 123456789 = 0x75bcd15.
- QOS traffic class is 4.
- KeyID is 2.
- Source MAC address is 0x123456789abc

- Destination MAC address is 0x23456789abcd

AES-OCB encryption is performed as follows:

Key	00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
Nonce	75 bc d1 54 12 34 56 78 9a bc 23 45 67 89 ab cd
Plaintext	aa aa 03 00 00 00 08 00 45 00 00 4e 66 1a 00 00 80 11 be 64 0a 00 01 22 0a ff ff ff 00 89 00 89 00 3a 00 00 80 a6 01 10 00 01 00 00 00 00 00 00 20 45 43 45 4a 45 48 45 43 46 43 45 50 46 45 45 49 45 46 46 43 43 41 43 41 43 41 43 41 43 41 41 41 00 00 20 00 01
Ciphertext	3f 42 a7 b6 8c 4d ea 07 e1 2b 2d c6 82 e4 72 81 70 45 a3 02 21 ee b6 bf 0a f0 26 a6 8d 73 e6 f3 55 dc 15 60 8e 7e 92 52 0b 4e 11 73 a3 a5 ce 7c 39 93 65 70 a1 16 40 79 02 a1 85 17 f6 5e 9d eb 35 bf 2c 18 49 6c 2c 2b 33 8b d4 0f d5 4e 1c 50 eb fd fa 56 95 56
Tag	06 9e e2 41 c6 1e 60 2d b3 05 76 53 03 a5 5f 5e

The first three octets of the nonce are the upper 24 bits of the replay counter value. The upper nibble of the fourth octet of the nonce consists of the least significant 4 bits of the replay counter value. The lower nibble of the fourth octet of the nonce is the QOS traffic class. Octets five through ten of the nonce are the source MAC address. Octets eleven through sixteen of the nonce are the destination MAC address. The plaintext consists of the MSDU data.

The expanded MSDU, after AES-OCB encapsulation, is as follows:

Replay	75 bc d1 85
MSDU data	3f 42 a7 b6 8c 4d ea 07 e1 2b 2d c6 82 e4 72 81 70 45 a3 02 21 ee b6 bf 0a f0 26 a6 8d 73 e6 f3 55 dc 15 60 8e 7e 92 52 0b 4e 11 73 a3 a5 ce 7c 39 93 65 70 a1 16 40 79 02 a1 85 17 f6 5e 9d eb 35 bf 2c 18 49 6c 2c 2b 33 8b d4 0f d5 4e 1c 50 eb fd fa 56 95 56
MIC	06 9e e2 41 c6 1e 60 2d

The first three octets of the replay field are the upper 24 bits of the replay counter value. The fourth octet of the replay field is the concatenation of: the 2-bit keyID value; two 0-bits; and the least significant 4 bits of the replay counter value. The MSDU data consists of the ciphertext. The MIC is the first eight octets of the tag value.

F.10.5 The PRF Function - PRF(key, prefix, data, length).

A set of test vectors are provided for each size of PRF function used in this specification. The input to the PRF function are strings for ‘key’, ‘prefix’ and ‘data’. The length can be any multiple of 8, but the values: 192, 256, 384, 512 and 768 are used in this specification. The test vectors were taken from RFC2202 with additional vectors added to test larger key and data sizes.

Test_case	1
key	0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b
prefix	"prefix"
data	"Hi There"
length	192
PRF-192	bc d4 c6 50 b3 0b 96 84 95 18 29 e0 d7 5f 9d 54 b8 62 17 5e d9 f0 06 06

Test_case	2
key	'Jefe'
prefix	"prefix-2"
data	"what do ya want for nothing?"
length	256
PRF-256	47 c4 90 8e 30 c9 47 52 1a d2 0b e9 05 34 50 ec be a2 3d 3a a6 04 b7 73 26 d8 b3 82 5f f7 47 5c

Test_case	3
key	aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa

	aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
prefix	"prefix-3"
data	"Test Using Larger Than Block-Size Key - Hash Key First"
length	384
PRF-384	0a b6 c3 3c cf 70 d0 d7 36 f4 b0 4c 8a 73 73 25 55 11 ab c5 07 37 13 16 3b d0 b8 c9 ee b7 e1 95 6f a0 66 82 0a 73 dd ee 3f 6d 3b d4 07 e0 68 2a

Test_case	4
key	0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b
prefix	"prefix-4"
data	"Hi There Again"
length	512
PRF-512	24 8c fb c5 32 ab 38 ff a4 83 c8 a2 e4 0b f1 70 eb 54 2a 2e 09 16 d7 bf 6d 97 da 2c 4c 5c a8 77 73 6c 53 a6 5b 03 fa 4b 37 45 ce 76 13 f6 ad 68 e0 e4 a7 98 b7 cf 69 1c 96 17 6f d6 34 a5 9a 49

Test_case	5
key	aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
prefix	"prefix-5"
data	"Test Using Larger Than Block-Size Key and Larger Than One Block-Size Data"
length	768
PRF-768	67 27 a3 e8 d5 2c f2 70 08 ce 4d 68 3e 45 99 25 c6 23 5b e0 0c 8c 13 03 77 26 af fc bc 02 29 17 a5 94 1c 0c 77 4b 00 25 7f 77 c6 e2 4c 81 02 87 8e 04 b7 2c f6 c7 88 a7 ba ec 4f 69 68 7b eb d6 30 15 59 ca 1f c2 6f 93 04 2e 1e 82 ba 28 9a 05 2c a8 51 ef cd 4e 15 a1 5d d0 4c bb e1 f6 94 58

F.10 Key hierarchy test vectors

The following test vectors provide an example of both pairwise and group key derivation for CCMP, TKIP and WRAP.

F.10.1 Pairwise Key Derivation

Pairwise keys are derived from the PMK, AA, SA, SNonce and ANonce. The PMK in this example is taken from the phasephrase generation test vector with pkbkdf2("ThisIsAPassword", "ThisIsASSID" ,4096 ,256). The values below are used for as input to the pairwise key derivation test vectors.

PMK	0d c0 d6 eb 90 55 5e d6 41 97 56 b9 a1 5e c3 e3 20 9b 63 df 70 7d d5 08 d1 45 81 f8 98 27 21 af
AA	a0 a1 a1 a3 a4 a5
SA	b0 b1 b2 b3 b4 b5
SNonce	c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 d0 d1 d2 d3 d4 d5 d6 d7 d8 d9
ANonce	e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 f0 f1 f2 f3 f4 f5 f6 f7 f8 f9

F.10.1.1 CCMP Pairwuse Key Derivation

Using the values from section F.x.1 for PMK, AA, SA, SNonce and ANonce the key derivation process for CCMP generates:

TK1	8c b7 78 33 2e 94 ac a6 d3 0b 89 cb e8 2a 9c a9

F.10.1.2 TKIP Pairwise Key Derivation

Using the values from section F.x.1 for PMK, AA, SA, SNonce and ANonce the key derivation process for TKIP generates:

MK	aa 7c fc 85 60 25 1e 4b c6 87 e0 cb 8d 29 83 63
EK	ba 53 16 3d f3 2a 86 38 f4 79 ab e3 4b fd 2b c8
TK1	8c b7 78 33 2e 94 ac a6 d3 0b 89 cb e8 2a 9c a9
TK2	36 4a ff bb ce 87 5f 5d f2 dd 58 41 c0 ed 2a 41
small_to_large_MIC_key	36 4a ff bb ce 87 5f 5d
large_to_small_MIC_key	f2 dd 58 41 c0 ed 2a 41

F.10.1.3 WRAP Pairwise Key Derivation

Using the values from section F.x.1 for PMK, AA, SA, SNonce and ANonce the key derivation process for WRAP generates:

TK1	8c b7 78 33 2e 94 ac a6 d3 0b 89 cb e8 2a 9c a9

F.10.2 Group Key Derivation

Group keys are derived from the values of GMK, AA and GNonce. The test vectors in the following sections use these values to generate the associated group keys for CCMP, TKIP and WRAP.

GMK	01 23 58 13 21 34 55 89 14 42 33 37 76 10 98 71 59 72 58 44 18 16 76 51 09 46 17 71 12 86 57 46
AA	a0 a1 a1 a3 a4 a5
GNonce	00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13

F.10.2.1 CCMP Group Key Derivation

Using the values from section F.x.2 for GMK, AA, and GNonce the key derivation process for CCMP generates:

TK1	02 36 05 a1 ae 5b e4 d1 ba b4 7e 40 2f a4 da 5e

F.10.2.2 TKIP Group Key Derivation

Using the values from section F.x.2 for GMK, AA, and GNonce the key derivation process for TKIP generates:

TK1	02 36 05 a1 ae 5b e4 d1 ba b4 7e 40 2f a4 da 5e
TK2	65 73 96 c6 c6 de bc 5f 67 fc 80 bf 9a be ce 25
small_to_large_MIC_key	65 73 96 c6 c6 de bc 5f
large_to_small_MIC_key	67 fc 80 bf 9a be ce 25

F.10.2.3 WRAP Group Key Derivation

Using the values from section F.x.2 for GMK, AA, and GNonce the key derivation process for WRAP generates:

TK1	02 36 05 a1 ae 5b e4 d1 ba b4 7e 40 2f a4 da 5e